

# A Programming Model for Precise Computation Control satisfying Application specific semantics

*Jayanth S Venkataraman*

Submitted to the Department of Electrical Engineering &

Computer Science and the Faculty of the Graduate School  
of the University of Kansas in partial fulfillment of  
the requirements for the degree of Master's of Science

## Thesis Committee:

---

Douglas Niehaus, EECS Department

---

Arvin Agah, EECS Department

---

Prasad Kulkarni, EECS Department

---

Date Defended

© 2008 Jayanth S Venkataraman  
All Rights Reserved

The Thesis Committee for Jayanth S Venkataraman certifies  
That this is the approved version of the following thesis:

**A Programming Model for Precise Computation Control satisfying  
Application specific semantics**

Committee:

---

Douglas Niehaus, EECS Department

---

Arvin Agah,EECS Department

---

Prasad Kulkarni, EECS Department

---

Date Approved

*To my parents and my brother, my biggest support*

## Acknowledgments

I would like to thank Douglas Niehaus, EECS Department, my advisor and committee chair, for providing guidance during the course of the thesis. I would also like to thank Arvin Agah, EECS Department and Prasad Kulkarni, EECS Department for serving as members of my thesis committee.

I would like to thank my parents for providing me all the infrastructure and moral support to study in the US. The biggest thanks goes to my brother, Bharath, who has been a pillar of strength for me right from the day i started preparing for my GRE.

The friends at Colony Woods, where i stayed, were a huge boost and served as a great relaxation during times of stress. Thanks to Prashanth, Vinay, Kannan, Muthu, Kanagaraj and others.

## **Abstract**

Many applications require specialized semantics for precise computation control of its components. The standard Linux kernel does not provide this since application semantics do not always map onto priority based scheduling and also because when a system is under high load, applications can be affected by the high CPU contention which makes the scheduler to not choose a particular application. This thesis proposes a customized programming model that provides a flexibly configurable scheduling framework for expressing application semantics as the solution. This programming model is called the State and Control variable programming model (SCPM). State and Control variable programming model (SCPM) allows the representation of the application threads and its semantics as part of the scheduling algorithm. In order to demonstrate such a programming model, this thesis uses a video pipeline application as its driving application.

# Contents

<b>Acceptance Page</b>	<b>2</b>
<b>Abstract</b>	<b>1</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Related Work</b>	<b>5</b>
2.1 Ingo Molnar’s Real Time Patch . . . . .	6
2.2 Completely Fair Scheduler . . . . .	8
2.3 Containers . . . . .	9
2.4 A Programming Model for Time-Synchronized Distributed Real-Time Systems (TSDRTS) . . . . .	9
<b>3 Implementation</b>	<b>11</b>
3.1 Implementation Environment . . . . .	15
3.1.1 CCF Framework and Group Scheduling . . . . .	15
3.1.2 Datastreams . . . . .	17
3.1.3 Taskmapper Framework . . . . .	19
3.1.4 KU Image Processing Library . . . . .	21
3.2 Components of the Solution . . . . .	22
3.2.1 State and Control Variable Programming Model . . . . .	23
3.2.2 Data Transfer Methods . . . . .	24
3.2.3 Scheduling Decision Functions . . . . .	26
3.2.4 Groupmaker . . . . .	27
3.2.5 Custom Application Specific Programming Model Components . . . . .	28

3.3	Interaction of the Components of the Solution . . . . .	30
<b>4</b>	<b>Experiment Design</b>	<b>47</b>
4.1	Applications . . . . .	48
4.1.1	Timed Playback . . . . .	49
4.1.2	Transformation Pipeline . . . . .	50
4.2	Schedulers . . . . .	50
4.3	Metrics . . . . .	57
4.4	Experimental Conditions . . . . .	58
<b>5</b>	<b>Experimental Results</b>	<b>61</b>
5.1	Timed Playback . . . . .	62
5.1.1	Lightly Loaded Conditions . . . . .	63
5.1.2	Heavily Loaded Conditions . . . . .	71
5.2	Transformation Pipeline Application . . . . .	79
5.2.1	Lightly Loaded Conditions . . . . .	80
5.2.2	Highly Loaded Conditions . . . . .	81
<b>6</b>	<b>Conclusions and Future Work</b>	<b>89</b>
	<b>References</b>	<b>91</b>



# List of Tables

# List of Figures

3.1	Generic Group Scheduling Framework . . . . .	16
3.2	Abstract Software Architecture . . . . .	31
3.3	Timed Playback Experiment GS Hierarchy and Pipeline Diagram	33
3.4	Timed Playback Application Software Architecture . . . . .	34
3.5	Timed Playback experiment SDF-Application interaction . . . . .	40
4.1	Timed Playback Application Pipeline Structure . . . . .	50
4.2	Transformation Pipeline Application Pipeline Structure . . . . .	51
4.3	Timed Playback Group Scheduling Hierarchy with RD Scheduler .	53
4.4	Timed Playback Group Scheduling Hierarchy under RDX Scheduler	55
5.1	FDTE-S under Self Timed with Light Load (FDTES-ST-UL) (A)High Level View(HLV) (B) Detailed View(DV) . . . . .	64
5.2	FDTE-S under RD Control with Light Load (FDTES-RD-UL) (A)High Level View (HLV) (B) Detailed View(DV) . . . . .	65
5.3	FDTE-S under RDX Control with Light Load (FDTES-RDX-UL) (A)High Level View(HLV) (B) Detailed View(DV) . . . . .	66
5.4	FDTE-C under Self Timed with Light Load (FDTEC-ST-UL) (A)High Level View(HLV) (B) Detailed View(DV) . . . . .	68
5.5	FDTE-C under RD Control with Light Load (FDTEC-RD-UL) (A)High Level View(HLV) (B) Detailed View(DV) . . . . .	69
5.6	FDTE-C under RDX Control with Light Load (FDTEC-RDX-UL) (A)High Level View(HLV) (B) Detailed View(DV) . . . . .	70
5.7	FDTE-S under Self Timed with High Load (FDTES-ST-HL) (A)High Level View(HLV) (B) Detailed View(DV) . . . . .	72

5.8	FDTE-S under RD Control with High Load (FDTES-RD-HL) (A)High Level View(HLV) (B) Detailed View(DV) . . . . .	73
5.9	FDTE-S under RDX Control with High Load (FDTES-RDX-HL) (A)High Level View (HLV) (B) Detailed View (DV) . . . . .	74
5.10	FDTE-C under Self Timed Control with High Load (FDTEC-ST-HL) (A)High Level View (HLV) (B) Detailed View(DV) . . . . .	76
5.11	FDTE-C under Read and Display thread Control with High Load (FDTEC-RD-HL) (A)High Level View(HLV) (B) Detailed View(DV)	77
5.12	FDTE-C under Read, Display and Xserver thread Control with High Load (FDTEC-RDX-HL) (A)High Level View(HLV) (B) Detailed View(DV) . . . . .	78
5.13	FDTE-S under Self Timed with High Load (FDTES-ST-HL) (A)High Level View(HLV) (B) Detailed View(DV) . . . . .	82
5.14	High Level View(HLV) of FDTE-S under RTDX Control with High Load (FDTES-RTDX-HL) . . . . .	83
5.15	Detailed View of FDTE-S with High Load under (A)RTD Control (FDTES-RTD-HL) (B) RTDX Control (FDTES-RTDX-HL) . . .	84
5.16	FDTE-C under Self Timed with High Load (FDTEC-ST-HL) (A)High Level View(HLV) (B) Detailed View(DV) . . . . .	85
5.17	High Level View(HLV) of FDTE-C under RTDX Control with High Load (FDTEC-RTDX-HL) . . . . .	86
5.18	Detailed View of FDTE-C with High Load under (A)RTD Control (FDTEC-RTD-HL) (B) RTDX Control (FDTEC-RTDX-HL) . . .	87

# List of Programs

3.1	Timed Playback Read from File Thread Application Work Loop .	39
3.2	Timed Playback Display Thread Application Work Loop . . . . .	40
3.3	Explicit Plan Video SDF Algorithm . . . . .	46
4.1	Self Timed Code Structure . . . . .	52
4.2	Self Timed Nanosleep time calculation function . . . . .	53
4.3	Read and Display Code Structure . . . . .	54
4.4	Read, Display and Xserver Code Structure . . . . .	56
4.5	FDTE Metric measuring Datastream events in the work loop . . .	58

# Chapter 1

## Introduction

Many applications require a level of precision in system scheduling that is not provided by the Standard Linux scheduler due to the following issues. Application semantics for real time and other applications do not always map directly onto priority based scheduling which is the standard interface provided by most operating systems. Share based or other scheduling approaches suffer the same limitation because a single scheduling semantic is never going to be adequate for all applications. Thus a flexibly configurable scheduling framework must be provided by systems so that application developers can implement customized programming models including customized schedulers when the semantics of their applications require them.

Applications can be affected by the load on the system. Load being characterized by the number of high priority and other processes, the number of operating system computation components (Hard IRQ's, Soft IRQ's and Tasklets) executed outside of the thread scheduler's control in service of disk activities, network traffic, additional operating system administrative duties in the system, etc.

Such a framework should also facilitate placing all application and system com-

putation components under unified control to ensure that the desired behaviour can be produced. Existing efforts in Standard linux and even in Ingo Molnar's Real Time(RT) patch [5] do not permit this and some extension to the system is required to solve this problem.

No mechanism is available in Standard linux to provide a representation of application structure and execution behavior semantics to the system and to how those semantics enforced by the scheduling subsystem. The Standard linux scheduling framework does not permit combining the representation of application semantics with a representation of OS computation components that support the application semantics as well. A fully integrated solution to scheduling both application semantics and Operating System computation components supporting the application is required to achieve precise control over application behavior.

Ingo Molnar's RT Patch [5] provides a kernel thread context to OS computation components like Hard IRQ's, Soft IRQ's and tasklets. Providing thread context for IRQ's help by bringing OS computation components into the thread scheduler's domain to an extent but it does not address the problem of fully representing the application semantics and all components supporting the computation. Note that the OS computation component threads are controlled by the standard Linux RT scheduler.

The RT patch [5] provides preemptible semaphores but these do not solve the problem of integrating concurrency control and scheduling semantics which is necessary for a completely general solution but not necessary for the work described here. However, the work described here does require implementing a customized scheduling algorithm for a specific set of threads which is also not supported by Ingo Molnar's RT patch [5].

Ingo's Completely Fair Scheduler (CFS) [7] is a CPU share scheduler and suffers the same problem as the Standard Linux priority based scheduler which cannot represent the arbitrary application semantics. The CFS [7] framework provides for stacking scheduler's and thus providing additional semantics. Group scheduling is an existing framework supporting flexible and application aware scheduling semantics which can be plugged into the CFS framework. Note that arbitrary scheduling semantics can be implemented under Group Scheduling and thus no further stacking within the scheduler stack is necessary.

Containers [4] are a way of describing CPU and other resource partitions assigned to groups of threads but these efforts generally do not provide the precise computation control under arbitrary scheduling semantics that Group Scheduling does. We anticipate that our Group Scheduling work and issues addressed by container work will converge in the future.

Developers who need about application specific scheduling semantics needs more precise control of computation behavior than is provided by the Standard Linux scheduler or is possible under the RT patch. Both application specific and more precise computation control is possible using methods developed in the KU Group scheduling effort in general and the extensions of that work described in this thesis. The topic of this thesis is a programming model implemented within the Group Scheduling framework that permits representation and enforcement of a wide range of application semantics.

The Related work chapter places the contribution of this thesis in the context of related work in greater detail. The implementation chapter first describes the relevant aspects of the KU System programming (KUSP) platform (Datastreams and Group Scheduling) and then describes the implementation of the State/Control

Variable Programming Model (SCPM) which provides flexible support for (1) specifying a wide range of application semantics and (2) implementation of precise control implementing the specified semantics. The Experimental Design chapter describes the experiments that were conducted to demonstrate the customized programming model, the experimental conditions under which it was tested and the metrics that need to be taken into consideration during the evaluation of the programming model. The Evaluation chapter discusses the evaluation of results that were obtained from running the experiments under the various conditions. The conclusion chapter describes the various inferences from the customized programming model and some of the future work that can be done.



# Chapter 2

## Related Work

Our goal in building a customized programming model is to provide a framework by which systems can exert greater control over how applications execute to satisfy their specified semantics. This control is achieved by allowing the applications to express their semantics to the system scheduler which is a custom scheduling mechanism in our programming model. The problem that we are addressing in this work is a general problem that a lot of real time and other applications face. So, there have been efforts previously which have addressed similar problems. These efforts are do not map directly onto expressing application semantics to the system scheduler but they are designed to improve the response time of real time applications. We will be discussing such efforts that are related to this thesis work in this chapter.

Ingo Molnar's RT Patch to the standard Linux Kernel provides kernel thread execution context for OS computation components which provides greater control of the system by the system scheduler. Section 2.1 will talk about the RT Patch in more detail.

The Completely Fair Scheduler (CFS) [7] is the new standard scheduler that

has been introduced in the Linux Kernel. It handles CPU resource allocation for executing processes and aims to maximize overall CPU utilization while maximizing interactive performance. Along with this scheduler comes the scheduler stack that has been introduced in the 2.6.23 Linux Kernel. The CFS is the based scheduler stack and other scheduling policies can be stacked on top of it. This is discussed in the Section 2.2.

Partitioning system resources using Containers [4] is an interesting discussion that is analogous in some ways to the problem discussed in the work. Containers can track and charge utilization of system resources like memory, tasks, etc. This will help administrators to utilize the available resources in a more efficient manner. The Section 2.3 discusses this in greater detail.

A Programming Model for Time-Synchronized Distributed Real-Time Systems (TSDRTS) [8] is a theoretical Programming Model that extends certain constraints on the system that can be easily mapped on to the Customized Programming Model that the work in this thesis discusses. Section 2.4 explains this.

## **2.1 Ingo Molnar's Real Time Patch**

Ingo Molnar is a Linux Kernel developer who has created a series of patches to the Standard Linux Kernel through which Real time efficiency is made available to the Linux Kernel. The patch that he creates is called the RT (Real Time) patch and a paper on the internals of his work has been provided in [5].

The RT patch [5] brings the OS computation components like Hard IRQ's, Soft IRQ's and Tasklets under the thread model by providing a kernel thread execution context for each type of Hard IRQ and Soft IRQ. These OS computation components were previously being executed in the current application

thread's context. For the analysis of a real time application, we would require the behaviour of applications to be predictable which is not possible when the OS computation components are running under the context of randomly selected application threads. This is because the OS computation components will eat into the amount of time available to the application thread and thus it is conceivable that it will reduce the performance of real time applications. Thus, the threadification of these components is an advantage to the Linux Kernel.

When the Standard Linux scheduler is used along with the RT patch [5], the OS computation component threads are placed under the real time scheduling domain and this reproduces the Standard Linux scheduler semantics. Thus the same semantics of the Standard Linux scheduler is also applied to these component threads. Under Group Scheduling as described in Section 3.1.1, it is possible to describe other kinds of scheduling semantics to these components. This will help the application in configuring not only the behaviour of its threads using Group Scheduling but also the behaviour of OS computation component threads. This provides the flexibility for real time applications to control the application more precisely. This kind of control has been done by the KUSP group but the work in this thesis did not require its use. The work described in this thesis concentrated on providing customized scheduling semantics for the application-level threads.

Preemptible Semaphores brings about the concept of Priority Inversion for higher priority tasks that are waiting for semaphores that are held by lower priority tasks. Preemptible Semaphores makes preemption finer grained which in general improves the Real Time performance. This is Ingo Molnar's motivation but this is useful in more general ways to increase the precision with which program behaviour can be controlled whether the semantics desired are real time or

something else.

## 2.2 Completely Fair Scheduler

The Completely Fair Scheduler (CFS) [7] is the name of the OS scheduler which was introduced into the 2.6.23 release of the Standard Linux Kernel. It handles CPU resource allocation for executing processes and aims to maximize overall CPU utilization while maximizing interactive performance. It was designed and implemented by Ingo Molnar. The CFS scheduler [7] replaces the previously used  $O(1)$  scheduler and uses a red black tree that implements a timeline for future task execution that tracks the CPU share used by each thread.

This effort also includes creating a scheduling stack framework in which the Real Time scheduler domain is loaded on top of the CFS [7]. This framework also provides support for other schedulers to be stacked on top of the Real Time scheduler. In the work described here the Group Scheduling framework is stacked on top of the Real Time scheduler which is on top of the basic CFS scheduler [7]. By stacking Group Scheduling on top of the stack, Group Scheduling has a chance to take decisions based on its hierarchy before the Real Time scheduler or the CFS [7] is consulted. The programming model customizing application semantics described in this thesis makes use of this mechanism for its implementation. In separate work that is not described in this thesis, Group Scheduling may replace the scheduler stack and RT and CFS [7] domains may become SDF's under the Group Scheduling model.

## 2.3 Containers

Containers [4] are a Google based effort to partition resource allocation for CPU and other resources in the system. This is giving opportunity to run different workloads on the same platform for better hardware resource utilization. Containers [4] will allow system admins to customize the underlying platform for different applications based on their performance and hardware resource utilization needs. Containers [4] interact with the CFS scheduler in order to provide information as to how much CPU share needs to be provided for different resources and how the allocation affects the scheduling in the system.

Containers [4] can conceivably converge with the Group Scheduling's view of groups of threads, the resources they use and the SDF's controlling how the threads use the resources. This is some future work that is possible. In this convergence, we could see that this is analogous to what our State and Control Variable Programming Model (SCPM) has set out to achieve. Allocation of resources according to user specification is similar to expressing application semantics to Group Scheduling. But, by using containers we cannot control the scheduling semantics of a system like the way we can do with our SCPM because containers do not support customized schedulers for each container.

## 2.4 A Programming Model for Time-Synchronized

### Distributed Real-Time Systems (TSDRTS)

TSDRTS's programming model [8] has a strong theoretical basis but the model cannot be adequately supported by the Standard Linux Kernel. The reason that this programming model cannot be support is because it requires a set of nodes

describing data flow to be executing predictably with respect to time, which is not guaranteed by the Standard Linux Kernel because of its delay in scheduling.

The problem described in the paper is similar to that of the problem addressed in this thesis work. The problem in this paper [8] describes a set of nodes that would form a multiple pipeline stages which are governed by certain rules that need to be followed in order for this programming model to be successful. The rule is that these pipeline stages must be time synchronized. That is, data that enters a pipeline must be processed by each node before a certain pre-determined time. This would fit in easily with the SCPM since the nodes that want to exhibit time synchronization must express their time based requirements as state variables and the SDF can also control the nodes using control variables. The SDF's semantics will also include sharing the CPU among the nodes since they need to be synchronized. This will be a problem that would be perfectly attainable by the SCPM. Also, the TSDRTS model will find it tough to express its semantics to the standard Linux scheduler as it is priority based, thus enhancing the usability of the SCPM.

# Chapter 3

## Implementation

Application behaviour semantics are sometimes difficult or impossible to express in the semantics provided by the standard Linux scheduler. The standard Linux scheduler's priority based scheduling algorithm does not directly map onto the application semantics of threads. When the system is under high load, the standard Linux scheduler makes choices that result in the application not behaving as they are supposed to. This is due to the enhanced contention for the processor by competing processes and the lack of direct representation of application semantics.

Thus more precise behaviour control and measurement methods are required for controlling applications. This chapter explains the various components involved in the implementation of a State and Control Variable Programming Model (SCPM) that can be used to provide precise computation control of applications.

The implementation environment on the kernel side is vital to such a programming model. The standard Linux Kernel has been modified to provide services that will aid in the implementation of the SCPM. We give an introduction to the components required for such an environment in the following paragraphs.

The Computation Control Framework (CCF) is an effort that allows flexibly configurable scheduling semantics to control threads that require them. Group Scheduling (GS) is a kernel module that is loaded on top of the CCF which allows applications to express its semantics as part of the system scheduling algorithm. GS views the application threads as a hierarchy that is composed of groups in which each group has threads as their members. Each of these groups is governed by a scheduling algorithm that is used to take decisions as to which thread from the group should be run by the CPU. This work is outside the scope of this thesis, but is discussed in more detail in Section 3.1.1.

Datastreams lets a user gather performance related information from the Linux Kernel and user space programs in a standard format. That enables the user to combine the two and derive meaningful views of application and system performance. This information also contain timestamps for when events occurred and some associated data of the event. These events from the kernel and user space can then be merged and postprocessed to create meaningful interpretations of the information. This is discussed more in detail in Section 3.1.2.

Representation of processes in the system is done using PID's which changes each time a program is executed and does not provide a consistent view of the application. The consistent view is necessary when many threads must co-operate to achieve a particular application behaviour. This has been achieved by the Task Mapper framework which is a kernel framework that keeps track of mappings between names and task ID's for each thread, which is consistent across program instances. This is discussed in more detail in Section 3.1.3.

The driving application that is used in all the experiments of this work is created by using the KU Image Processing (KUIM) library. This has been chosen



as the driving application because it can be used to construct pipelines of video processing nodes. Such pipelines bring to effect different constraints on application behaviour that are tough to enforce to when the semantics do not map to the scheduling algorithm used in the system. In our example, we would ensure that playback of the video follows a specific frame by frame schedule. This kind of constraint aids in proving our basic premise that our SCPM can perform better than the standard Linux scheduler based programming model. This is discussed in more detail in Section 3.1.4.

The SCPM contains different components that have been implemented on top of the implementation environment introduced in the preceding paragraphs. These components make the Programming Model to be customizable and flexibly configurable by the application developer. The components are introduced in the following paragraphs and will be discussed in detail in Section 3.2.

The SCPM is a method by which the application can transfer application state data variables to the Scheduling Decision Function (SDF) controlling the application and read control variables published by the SDF. This is discussed in the Section 3.2.1.

For the SDF's to obtain the state information from the application there needs to be data transfer methods available. One of them is by using the basic IOCTL system call and the other is by using shared memory. This is discussed in detail in the Section 3.2.2.

The SDF controlling the application are the core component of the SCPM and it is these which actually affect the way the application threads are scheduled in the system. Section 3.2.3 explains this in detail.

Every application has to setup its own GS hierarchy in order to use GS to

control their threads. In order to reduce the complexity of the setup process for GS, a Groupmaker utility has been used. This utility lets the user specify the GS hierarchy, its SDF's and other information in a human readable format that Groupmaker takes as input and uses to set up the GS hierarchy. This is explained in the Section 3.2.4.

In the experiments done for this thesis, we have used the Timed Playback(TP) application as the driving application. To bring this application under our SCPM, we needed some custom application specific utilities. One of them is an interfacing library between the KUIM application and the rest of the SCPM called the KUSP-KUIM utilities. The Frame Schedule utility is one which is used by the application to specify a frame playback display schedule to the SDF for the TP application. This interprets the frame schedule specification file and sends it to the Explicit Schedule utility, which converts the frame schedule to the data structure that the SDF can understand. All these components are discussed in detail in Section 3.2.5.

Section 3.3 discusses how the various components of the Programming Model interact with each other. First, it discusses the abstract software architecture of the TP using the SCPM. Then, the Timed Playback application and its interaction with the components specific to the application is discussed. The goal, the motives, the pipeline structure, the GS hierarchy used by the TP application is also discussed. The software architecture and the application's interaction with the SDF is also discussed. The scheduling algorithm that the TP application uses is the Explicit Plan Video SDF. Their semantic considerations and their algorithm is also stated. Section 3.3 will explain the work flow of the SCPM by using the Timed Playback application as its example.

### 3.1 Implementation Environment

The basic programming model that uses the standard Linux scheduler has a lot of disadvantages when it is controlling applications that require scheduling to be done according to their individual application semantics. This is because the basic Linux programming model does not have a method by which the application programmer can directly express and enforce the application semantics.

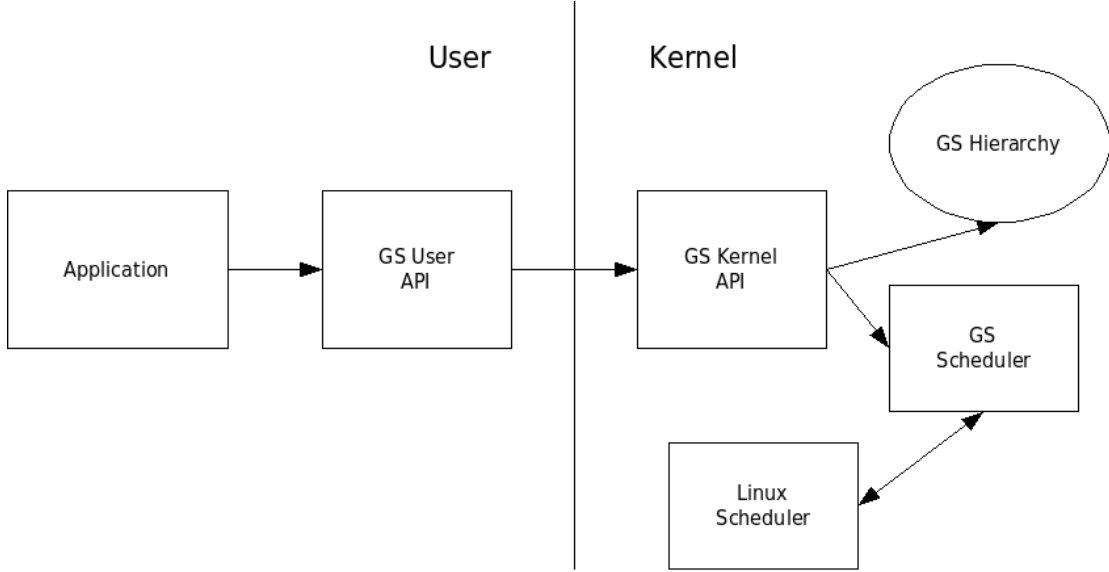
The environment to implement such a programming model contains CCF and GSfor providing flexible scheduling semantics implementation, Datastreams for gathering data and Taskmapper to map names to threads. These collectively form the Kernel environment used by the work described here. On the user side, KUIM is used as the application framework that helps to create video pipeline structures which can be a good testing ground for the proposed SCPM.

#### 3.1.1 CCF Framework and Group Scheduling

The Computation Control framework is one within which arbitrary scheduling semantics can be implemented. This framework has been developed and has been provided for the use of this thesis work. The papers [6] and [2] discuss about the need for such a framework, how it is implemented and what its implications are. The CCF framework is the patch to the Standard Linux Kernel while Group Scheduling is a kernel module that is loaded into the CCF framework which provides flexibly configurable scheduling semantics and programming model support. A customized programming model is created by implementing one or more GS SDF's to control application component execution plus associated user-level API and other support routines and tools.

The CCF framework patch to the Standard Linux Kernel consists the imple-

mentation of exclusive control of threads. Exclusive control guarantees that a particular thread, when it chooses to be exclusively scheduled only by CCF, is choosable only by the CCF framework and not by the standard Linux scheduler.



**Figure 3.1.** Generic Group Scheduling Framework

Figure 3.1 shows the generic GS framework. When the application decides upon the GS hierarchy that it is going to use, it makes a set of IOCTL system calls giving information to the GS kernel framework through the GS user API to setup the GS hierarchy and the SDF for all the groups in the hierarchy. The same system call method is used for other interaction between the application and the GS or the SDF kernel module.

Group Scheduling introduces a representation of computations in the form of a group hierarchy. It allows threads to join groups which are governed by a scheduling algorithm which is in the form of a Scheduling Decision Function(SDF). This scheduling algorithm is invoked from the system scheduler context when the group is part of the GS hierarchy. The CCF framework is called from the standard

Linux Kernel scheduler and the CCF framework calls the SDF at the top level group in the GS hierarchy to decide if a thread under GS control should be the next to use the CPU. This group can query another sub-ordinate group or can decide by itself depending upon the kind of scheduling algorithm. The SDF's are also loaded as kernel modules which contain a standard set of callback routines that are registered with the GS base module when it is loaded as a kernel module.

The SCPM that is described in this thesis uses this generic GS framework to control applications that need to be scheduled according to particular scheduling semantics. The SDF in GS is the decision making authority that is queried by the basic Linux scheduler to provide it a thread to run next. Since GS takes precedence over the Linux scheduler in decision making, control of applications according to a indigenous scheduling algorithm is achieved.

### **3.1.2 Datastreams**

Datastreams is a method of gathering performance related data from both the application and the kernel level which enables users to extract a detailed view of application behaviour and the system behaviours that affect it. Datastreams is a kernel module that is loaded on top of the Datastream hooks in the Linux Kernel which has modified the standard Linux Kernel in order to enable the use of Datastreams. It is a very important tool that is used in the gathering and representation of results in this thesis work.

Datastreams gathers information in the form of detailed timestamped event streams and aggregate data such as histograms. Each of these types of information will contain data associated with it that can be used to make inferences during analysis time. The timestamp of each of these events or histograms is produced

by using the CPU Time Stamp Counter (TSC) values. The TSC is a 64 bit register in all X86 based processors where during each clock signal the TSC value is incremented by one. This register can be read through an assembly language instruction and is available to the Kernel. So, if a processor is running at a clock speed of more than 1 GHz, then the register is incremented once every nanosecond. Thus TSC has a nanosecond resolution that is available to the event analyzer of Datastreams. Datastream information can be collected from user space as well as the kernel as separate streams and the two streams can be merged as well. This merging of different data streams is important in many analysis problems. Datastreams organizes its namespace in the form of families and multiple events or histograms come under a family. When a particular code segment wants to use Datastreams, it specifies the family name, the name of the event or histogram and the data associated with the event.

In the kernel, when we need to record data using Datastreams, we use the Datastreams Kernel Instrumentation (DSKI) Daemon which creates an event stream with buffers that can be used to store the events. The user has the option to name specific events that have to be logged in the form of a specification file that is an input to the DSKI Daemon. When an instrumentation point is encountered in the code, a Datastream macro is called from the code and it makes calls to log the generated event in the output buffer. This output buffer is periodically written on to an output file in binary format. The final output of Datastreams is a binary file of generated events.

This binary file is then used by the DS Postprocessing (DSPP) engine in order to create representations and interpretations of the gathered data. DSPP is a framework that has been developed by the KUSP group using Python. This

framework allows the user to write filters that would interpret the data as a stream of events and create inferences from it. The DSPP framework can look at the data associated with the events and the timestamps of the events. With all this information, the DSPP framework can create histograms for analysis of data. This kind of extensive postprocessing of the gathered data enables users to extract a wide variety of views of system and application behaviour which can be used to check that all application constraints are satisfied and for debugging when constraint violations occur.

### **3.1.3 Taskmapper Framework**

The Standard Linux Kernel views each system level thread as a separate entity. Application computations are often composed of multiple threads that must cooperate to achieve the desired application behaviour semantics. These applications will be better served if there was a consistent and aggregate method to represent applications in the Kernel. The current method of using PID's to represent threads in the kernel is not consistent since it changes for each instance of a process and also no aggregation is possible.

In order to have a constant representation of programs during different instances of their execution, we need to have a mapping from the name of the thread to the thread data structure so that kernel or user space code can refer to this thread by its name and in return get its task structure or PID. This is what the Taskmapper framework achieves. It is a kernel framework that contains data structures that keep track of name to task structure mappings. This framework gets a call from the user thread specifying its name and maps it to its corresponding task structure. There is also another variant of this wherein we can give a

single name to a group of threads.

The name and thread set associations are used to describe how sets of threads are managed by GS hierarchies and to describe datastream performance information gathering in many cases.

An important part of TM is that the definition of the namespace elements and their association with specific threads is flexible with respect to thread creation times and scheduling or data gathering definitions. The effect of this is that GS hierarchies can be defined before the threads being controlled exist without confusion because the Taskmapper, Group Scheduling and Datastreams frameworks co-operate to correctly handle the situation. This is referred to as Taskmapper's rendezvous with Group Scheduling and Datastream. Below, we show the sequence of events that take place during the rendezvous between Taskmapper and Group Scheduling.

1. GS is called to add a thread as a member in a group by providing the member a name.
2. GS makes a call to the Taskmapper Kernel library to check if it contains any mapping for the name provided.

If Yes, then Taskmapper returns the task struct to the GS framework and the member is added as a member into the group.

If No, then Group Scheduling registers a callback with Taskmapper, which contains a capability to store a callback function for a name, asking TM to notify it when Taskmapper gets a mapping between the name provided and a task structure.

3. At a later time, when the thread is created and it calls Taskmapper to reg-



ister its name, the Taskmapper framework finds out that a callback routine has been saved for this name and that it should call that function and notify it of the new mapping. Now GS knows of this name and its task structure and adds the thread as a member to the group that it was meant to be in.

### **3.1.4 KU Image Processing Library**

The KUIM pipeline is a framework that lets the user create different video pipeline structures that can bring about different video characteristics and help to show different behavioural constraints for running these pipelines at the application level. Each pipeline stage that is constructed in the experiment creates a separate thread. Multiple pipelines can be constructed as we can split the outputs of a pipeline. Each of these pipeline stages do processing on a single frame at a time by considering each frame as a color image. The KUIM has capabilities such as reading from a video file, reading from a camera and from the network through a socket.

The Standard Linux Kernel views the several threads comprising a KUIM pipeline as separate components and can only consider them individually and thus does not consider the application level semantics related to video playback. The application semantics of each of these pipelines can be varied and would impose constraints on the system that cannot be expressed to the Standard Linux scheduler. Using user level timers and thread state manipulations to try to achieve the desired behaviour is helpful but is not sufficient in achieving the desired semantics.

An application that has been created for this thesis work that demonstrates the above constraints is the Timed Playback application. The goal of this application is to play video precisely according to a pre-determined frame schedule. This

application contains 2 threads at its basic structure has one thread of them reading frames from a file and the other sending the frame to the Xserver to display the frame on the screen. KUIM Pipeline stages are connected with each other through a shared queue data structure. A KUIM queue is a shared object that connects 2 pipeline stages. It is an implementation of a First in First out (FIFO) queue with concurrency control using Pthread mutexes. The KUIM queue is a method at the user level to enforce the desired application semantics along with the usage of user level timers to achieve its semantics.

The Timed Playback experiments were run under the Standard Linux scheduler based programming model by using nanosleep as the method of implementing the frame schedule. When there is light load on the system, we could see that the desired effect was seen on the screen but under higher load, the video was displayed with a lot of jitter and the frame display diverged from the specified schedule significantly. These experiments are called the Self Timed experiments. This is discussed in more detail in Section 4.2. These experiments demonstrate that the self timing methods are not sufficient under all system load conditions and that the SCPM described in this thesis works well.

## 3.2 Components of the Solution

In Section 3.1 we discussed what kind of backbone framework is required and is in place in order to build a customized programming model. Also, there are modifications to the backbone structure such that this programming model can work efficiently. So the current section is a discussion of what kind of components were built on top of the backbone framework, what it means to the programming model and why it is constructed in that manner.

### 3.2.1 State and Control Variable Programming Model

The standard Linux based programming model is a method that lacks the ability to precisely control all applications according to their semantics since it is priority based. An application like timed video playback which requires precise timing according to a frame schedule, priorities do not map well on to the application's semantics. So, in order to address problems of this nature, our SCPM supports direct expression of application semantics in the system scheduling domain. We know from Section 3.1.1 that Group Scheduling is a system scheduling framework that can ensure complete control of specific threads that are governed by a SDF.

Direct implementation of the application behaviour semantics as a SDF often requires interaction between the SDF and the application to gather data about the application state and to specify any application control parameters. The approach described here supports arbitrary sets of variables describing application state and other sets of variables controlling application behaviour to enable implementation of SDF's with a wide variety of application aware semantics.

State variables are a set of variables or data structures that keep track of specific attributes of the application that are used by the SDF controlling application execution. These attributes can be any information that is useful for in the SDF decision making. The decision on what these state variables should be is left to the application programmer. The state variables are data that are directly relayed from the user level application to the SDF and represent the current state of the application. In the Timed Playback application that was briefly introduced in the Section 3.1.4, the number of frames produced by each stage of the pipeline is a state variable.

Control variables are another set of variables or data structures which are used to control the application and are set by the SDF. The values of these variables are computed in the SDF context and are used so as to control the execution behaviour of the application. The computation of these variables is done from information available about the application through kernel level data structures and from the state variables corresponding to that particular application. This data is then sent on request from the SDF to the application. Control variables are helpful as they tell the application in real time as to how their execution behaviour in the user space should be tuned in order to comply with the scheduling algorithm's decision making. In our Timed Playback application, having a high queue and low queue watermark were intended as control variables but in the experiments' execution, it turned out to not be necessary.

### **3.2.2 Data Transfer Methods**

Programming models which include SDF's which are aware of application state must have access to data describing the aspects of application state that influence scheduling decisions. This information is available in the user space but need to be communicated to the GS SDF that is controlling the application. Both user and kernel sides of the programming model must maintain data structures that are identical so that the data can be correctly interpreted when state variables are sent to and control variables are received from the GS domain. Further, there are 2 methods that are available for data transfer.

One of them is the basic system call `ioctl` method which passes a reference to the data which is then copied into the kernel address space and then the GS interprets that data to store in its data structures. In the SCPM, this system

call interface can also be used to notify the SDF of certain events which are not strictly state variables and for this purpose we can add `ioctl()` commands in the GS domain that can be used to send these notifications to the function in the SDF. The system call version also adds some amount of overhead on to the application but this overhead is insignificant when compared to the kind of performance that this Programming Model delivers which will be evident in the experimental results discussed in Chapter 5. When the system call is being made, concurrency control is used to ensure that state data remains inconsistent. The concurrency control primitive that is used is a spin lock on the group of which the application thread is a member. The reason for the spin lock is because the group maintains a list of members that execute on each CPU and no other member from this group on this CPU should be able to modify the data.

The other method that is used for data transfer is the shared memory method. This method creates a memory mapped region for each application thread where it can store the data structures pertaining to state and control variables. This memory mapped region is accessible from the user address space as well as the Kernel address space. The implementation of such a shared memory region uses the `mmap` system call to map the user memory region from user space onto the Kernel address space. The concurrency control for such a shared memory region is extremely important. There are two versions of the shared memory model. The first one was done by Keegan Flanigan as part of his work discussed in [1]. In this version, there were two copies of the state and control variables in the memory mapped region. This version kept state as to which copy was the current one and managed the concurrency by letting the publisher of data to write to the unused one in a form of double buffering. After this, using an atomic operation the state

of the memory mapped region is changed to make the new data the currently used version so that the latest information is viewed. The second version was implemented by Noah Watkins of the KUSP group. In this version, a shared spin lock is used to control data access. This is a spin lock that controls concurrency across the user-kernel boundary. This spin lock is held when either the application or GS wants to access the shared data. When it is held from the application side, it is a must that the application must be exclusively controlled by GS and GS to ensure that the lock holder has a exclusive access to the data uses its scheduling control of the application thread. Using shared memory will reduce the amount of time spent in making the system call to transfer the data from the user to the Kernel space.

### **3.2.3 Scheduling Decision Functions**

Scheduling decision functions are the most important component in the SCPM implementation. Each application can have its own set of SDF's which implement the application semantics that is required for the application to run according to the desired semantics. These SDF's implement the logic taking into account the state variables that the applications will provide during the execution of the application. Remember that these SDF's are called from the standard Linux scheduler's context and are instrumental in providing complete control of the application by GS.

Also, since GS is a multi level hierarchical setup, it is possible to have different SDF's for different groups with different logics that implement a certain application semantic. In this case, the construction of the group hierarchy to suit the application is extremely important.

The work described in this thesis involves implementation of SDF's associated with GS hierarchies implementing the somewhat complicated RD and RDX scheduling semantics which can be realized a few different ways with different sets of specific SDF's. The SDF semantics are discussed in detail in Chapter 4. Also, a scheduling algorithm for the driving application is discussed in Section 3.3.

### **3.2.4 Groupmaker**

Groupmaker is a user space utility that lets the application programmer to specify the GS hierarchy that he wants to set up for that application. Groupmaker uses a special purpose configuration language to specify the GS hierarchy and is implemented in Python. In this specification, we can describe the groups, their members, the SDF for each group, and the group hierarchy. We can also specify any group or member level data and their initial values.

The Groupmaker Python utility parses the specification file and creates a set of data structures that can be used by the application when it starts. All the application has to do is to call this utility and it will install the specified GS hierarchy. Also, Groupmaker makes use of the Taskmapper-GS rendezvous mechanism that was discussed in Section 3.1.3. This ensures the representation of threads can be done with names which eases the overall setup process.

During setup of the GS hierarchy, Groupmaker looks at the data structures and makes system calls to the Group Scheduling Kernel module and sets up the groups, their members, their corresponding SDF's and all the member and group level data.

### 3.2.5 Custom Application Specific Programming Model Components

The Timed Playback application of KUIM is used in this thesis as the driving application to test the SCPM. The SCPM requires interaction between the application and the SDF for it to work properly. To minimize the influence of the SCPM on the application software architecture, we have created some utilities that would be used by the application to interface with the rest of the SCPM.

The KUSP-KUIM utilities provide the interface that a KUIM application uses to talk to the SCPM components. The Frame Schedule utility and the Explicit Schedule utility are used as a pair to upload the frame schedule information to the Timed Playback application's SDF so that it can display frames according to Explicit Plan Video SDF.

#### KUSP-KUIM Utilities

This thesis describes the SCPM which is a programming model that will work for any application that requires such application aware scheduling semantics. In order to demonstrate this programming model, the KUIM library was used as the driving application since it had the capability to create video pipelines which can have many different characteristics. KUIM as a stand alone application has to be modified in order to interact with the SCPM. But this modification has to be small enough so that the source code of the application can have minimum changes in order to implement the SCPM.

So, in order to achieve this a separate KUSP(KU System Programs Group)-KUIM utilities library was written which contains all the code that the KUIM application requires to interact with the various components of the SCPM. This library contains functions that aid in updating state variables, reading control



variables, initializing the state and control data of a thread, calls to Taskmapper functions that can announce the name of application threads to the task mapping utility,etc.

### **Frame Schedule Utility**

A utility has been created in order to interpret the frame schedule that the user provides and convert it into data structures that can be interpreted in the kernel for its own usage. This utility is called the Frame Schedule utility.

The experiments that we have conducted to demonstrate this SCPM is a Timed Playback of Video experiment as discussed in Section 3.3. In the Timed Playback experiments, a video is associated with with a frame schedule that contains the values specifying to the delay between the display of successive frames on the screen. This frame schedule is then converted to the data structures in the Frame Schedule SDF that keeps track of the schedule and uses it to enforce the correct application behaviour. These kernel level data structures are then used by the SDF in its logic for different purposes which will be discussed in the Section 3.3.

The input to this utility is a file that contains the group that is going to the use the schedule, a name for the schedule and the schedule itself. The schedule is basically a set of numbers in milliseconds which depict the delay between successive frames on the screen. This file specification is written according to a set of rules which are specified by the utility creator and is being parsed by the Python based Configuration Language developed by the KUSP group. This Configuration language contains a parser that check for compliance with the language and convert it into data structures which can then be read from the frame schedule utility. Then it converts it into a format that is then recognizable by the utility

that interfaces with the SDF.

### **Explicit Schedule Utility**

The utility that interfaces with the SDF that the Timed Playback application uses is called the Explicit Schedule utility. This utility converts the output of the Frame Schedule utility to a format which specifies the relative start and stop times for each interval in the schedule. It is during this interval that the frame display is supposed to happen. This utility then sends the data structure containing the interval specification to the SDF that uses this, which in this case is the Explicit Plan Video SDF that will be discussed in the Section 3.3. This utility also has other functions that would start the Explicit Plan in the SDF, including a trigger to start the Explicit Plan that is specified by the application.

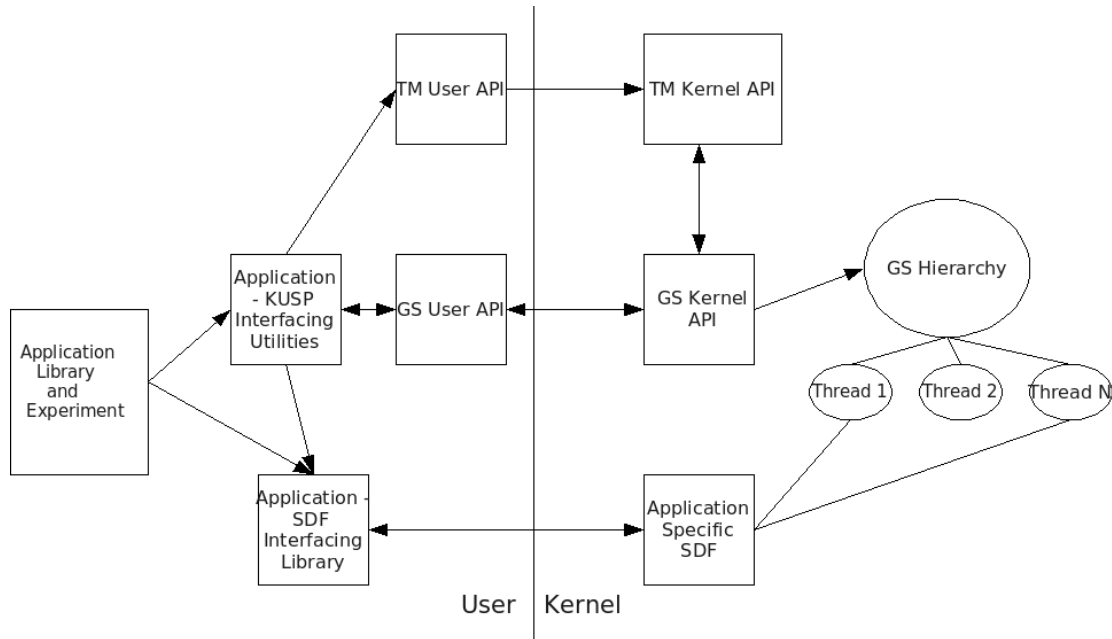
## **3.3 Interaction of the Components of the Solution**

In the previous sections we saw the various components that were built for the SCPM. These components help in various ways to create a TP application accurately controlling when video frames are displayed. Interaction of these components creates the SCPM and this is discussed in the current section. The SCPM is flexible to be used with any user application that requires it. So it is important to understand the abstract interaction of the components before going into a specific application as it would make it extensible to other applications as well.

### **Abstract Software Architecture of Programming Model**

The goal of the SCPM is to provide a framework that any application that wants to exhibit better control of its components can use. The abstract software

architecture of the SCPM is shown in Figure 3.2. This gives us the view of how the various components that were discussed in preceding sections of this chapter interact.



**Figure 3.2.** Abstract Software Architecture

In the figure you can see that an application that needs to be controlled under customized semantics is interfacing with multiple libraries which help in the various phases of application execution. One of the major advantages of having a configuration file that specifies the GS hierarchy is that the application itself need not worry about the GS setup. The Groupmaker application takes care of that aspect. It interprets the configuration file and makes the relevant calls to the GS Kernel API through the GS user api to setup the GS hierarchy. Depending on the application's thread availability at application start time, the GS rendezvous' with the Task Mapper kernel framework to resolve members with their respective names. The names that each thread in the application take should correspond to

the name in the GS specification file so that rendezvous between GS and Task Mapper representations can occur. The names of each of these threads are made known to the Task Mapper kernel framework through the Taskmapper user API.

In order to minimize the footprint of the KUSP changes required in the application, the KUSP-KUIM utilities library helps interfacing with the rest of the SCPM. This contains interfaces using which core actions like posting state variables, reading control variables, etc. can be done. If there are some actions that need to be specific to the SDF that controls the application, a separate user library that interfaces with the SDF in the Kernel is used. The goal of these interfaces is to make it easy to place a new application under SCPM control.

Each application must decide on the right semantics that are required for its purposes. The GS hierarchy that it requires must be set up and each group in the hierarchy will have an associated SDF. An application may require more than one group and SDF to achieve its semantic requirements.

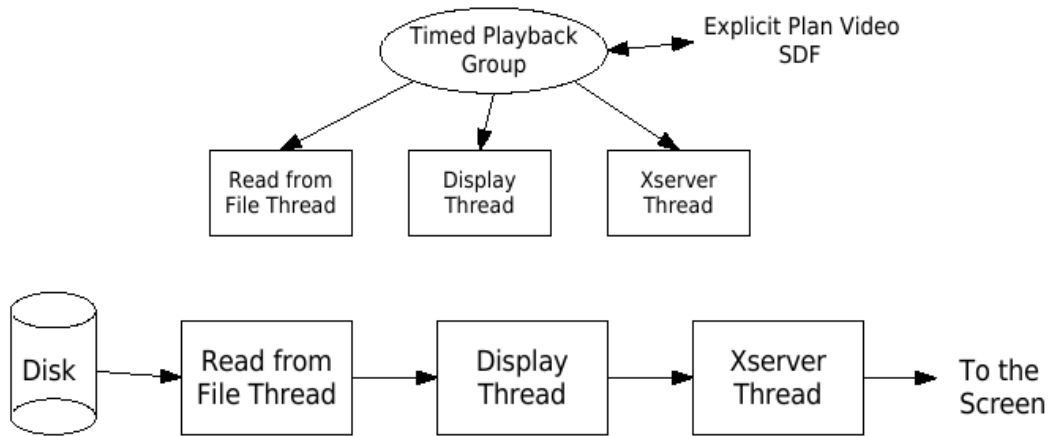
This is the abstract software architecture of the SCPM that can be used by any application that requires better control of its threads and it provides flexibly configurable components.

## **Timed Playback Application Implementation**

To demonstrate the interaction that happens between the various components of the SCPM specific to an application we will use the Timed Playback driving application.

The goal of the driving application is to create a video playback tool that will follow a set of frame delay and jitter values provided in the form of a frame schedule. This application has 2 different variants. The basic one has a Read from

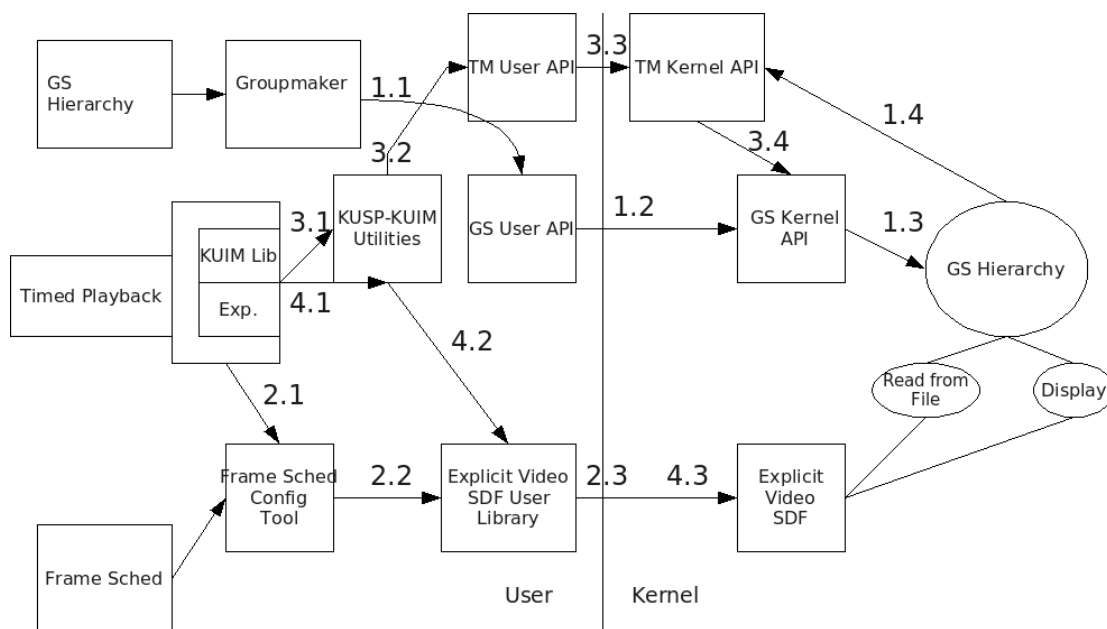
File thread and a display thread in the KUIIM domain. The other one, which we call the Transformation Pipeline, contains one or more transformation nodes in between the read and the display nodes that do some processing on the video. In both these variants the Xserver has been brought under GS control so that we can provide it a chance to display the frames that have been sent for display from the TP application.



**Figure 3.3.** Timed Playback Experiment GS Hierarchy and Pipeline Diagram

In Figure 3.3 we can see the first variant of the Timed Playback of video. The pipeline structure for this experiment is a single level pipeline in which the "Read from File" thread is the producer of frames from the file and the display thread sends the frame over to the xserver to display it on the screen. The Group Scheduling hierarchy for this experiment contains the 3 threads under the same group which is governed by the Explicit Plan Video SDF. The SDF that governs this group takes decisions based on the semantics of this application. The group that contains all the threads is the default group that will be queried by GS when the Standard Linux scheduler invokes it.

The Figure 3.4 shows the software architecture of the Programming Model with the interactions that take place between the various components of the Programming Model cutting across user and Kernel modes. In the Figure 3.4 we can see the sequence of events during the interaction of components. Below are the explanations of each of the interaction stages in numerical order.



**Figure 3.4.** Timed Playback Application Software Architecture

1. The Group Scheduling hierarchy specification is a file that is provided to the Groupmaker application. Groupmaker interprets the specification and makes the relevant GS calls to setup the GS hierarchy as shown from 1.1 to 1.3 by Figure 3.4. This call from Groupmaker is made even before the KUIM threads have been created by the application. GS will not be able to resolve these threads and thus it registers a notification request with Taskmapper to notify GS when the thread has been created as shown in 1.4 in same figure.
2. The Framesched utility was created such that the user can specify a set

of frame arrival times in the form of a schedule to the SCPM. The TP application invokes the Framesched utility as shown in 2.1 of Figure 3.4. The Framesched utility then interprets the frame schedule specification file provided by the user, sends it to the Explicit Plan Video SDF user library as an input data structure containing the schedule information. This is shown by step 2.2 of the figure. The Explicit Plan Video SDF then converts this data structure into a data structure that the SDF Kernel module can interpret. This is sent to the Kernel which stores the schedule in the format that it can best use. This is shown by step 2.3 of the figure.

3. The application starts off by spawning the relevant threads of this experiment. Each of these threads use the KUSP-KUIM utilities and the Task Mapper user API to publish its name to the Taskmapper Kernel API. This sequence is shown from 3.1 through 3.3 of Figure 3.4. During this process, the Taskmapper in the Kernel checks to see if it had received any notification requests for the threads in question. When it finds the requests, it calls the GS and notifies it of the existence of the threads. This is shown as step 3.4 in the figure. The GS now has members that have been resolved due to which it adds the members to the relevant groups.
4. The Display thread of the Timed Playback application calls the KUSP-KUIM utility to specify to the SDF that the explicit plan of the frame schedule can start as depicted as step 4.1 in Figure 3.4. This call is made through the Explicit Plan Video SDF user library which makes the IOCTL call to the SDF to start the explicit plan. These interactions are shown as steps 4.2 and 4.3 in the figure. With this call the frame schedule provided by the user is being imposed upon the application through the SDF.

Thus the software architecture of the SCPM is extremely important to understand the interaction and sequence of events that happen in the setup of the application with Group Scheduling. Once this setup has been done, the SDF controls the applications according to specified semantics. In order to understand the semantic details of the SDF, its goals and motives, the state variables that it uses to take its decisions, the sequence of interactions with the application at real time be discussed.

The goal of the Timed Playback application is to display the video according to a pre-determined frame schedule. In order to achieve this, the Display computation must be scheduled in a precisely timed fashion according to the frame schedule. Since this is a pipeline software architecture, the input queue of the Display thread must at all times contain frames which should be assured by the "Read from file" thread. Also, it is not sufficient that the Display computation be scheduled according to a frame schedule as it is the Xserver computation which actually displays each frame. This SDF brings under its control the Xserver thread which is provided a chance to run as soon as a frame has been sent to it from the KUIM application.

State variables are an integral part of the SCPM as it provides the state data in real time to the SDF for it to take its decisions. The Timed Playback application uses the following state variables to take its decisions.

**Hrtimers:** The explicit plan of the video playback is maintained by setting an execution interval of a pre-determined amount of time during which the SDF will choose the Display computation to run and display the frame. This execution interval is achieved by the use of hrtimers. The hrtimer is a high resolution timer provided by the Standard Linux Kernel. A timer goes off to start the execution



interval and another for denoting the end of the execution interval. Also, an execution interval within the Display thread execution interval is being used by the RDX scheduler to control the Xserver thread. This execution interval is also implemented using hrtimers.

**Current Display execution interval state:** This is a binary integer that shows whether the Display execution interval is active or inactive at any given time. This variable is used to decide when to choose the Display computation.

**Current Xserver execution interval state:** This is a binary integer that shows whether the Xserver execution interval is active or inactive at any given time.

**Number of display execution intervals:** This is a counter that is incremented each time a begin execution interval hrtimer is set. It is used by the SDF to compare with the number of frames produced by the display computation. This is important since this way we will know whether the display computation is actually keeping up with the frame schedule that it is supposed to be following.

**Number of frames displayed/completed:** This is a counter that is incremented each time the display thread completes a frame. This signifies that the application has completed the display schedule up to Frame 'X' and will be useful when keeping track of how well the application is adhering to the frame schedule.

**Input queue length:** KUIM uses a shared queue system where the queue is consumed by one thread and produced by another. So, one thread's input queue is another thread's output queue.

One of these variables tracks the number of frames that are available in the input queue of any pipeline stage thread. Also, the producer thread, in this case the "Read from File" thread, does not have an input queue and so its value is set

to -1.

**Output queue length:** One of these variables tracks the number of frames that are available in the output queue of any pipeline stage thread. Also, the consumer thread, in this case the Display thread, does not have a output queue since its output is to send each frame to the Xserver.

**Bootstrap Queue Length:** This is the minimum number of frames that need to be buffered in the output queue of the "Read from File" thread before the downstream threads in the pipeline can start processing the frames.

**Display Xserver Interaction:** This variable shows the various states that the interaction between the Display thread and the Xserver thread can be in. Line 5 through 7 of Program 3.2 shows the various states which the Display Xserver Interaction state variable takes with respect to the work loop of the Display thread.

We need this variable since there is a possibility that the Display thread blocks just after it sends the frame to the Xserver. If this happens then there needs an indication in SDF for it know the reason for the blocking. So, for this purpose this state variable is in the "About to Submit" state. And the "Submit Finished" is when the frame has been submitted and the SDF needs to provide a chance for the Xserver to run. In all the other places of the SDF work loop this variable takes a value of 0 or "No Interaction".

This set of state variables are used by the TP application to enforce the desired TP application semantics. Most of these state variables are being updated in real time from the application using the SCPM API. These interactions between the application and the SDF happen asynchronously to when the SDF runs. So, their interactions will have to be explained in order to fully understand the implementation. These interactions happen during the processing of each frame by the TP

application threads which we refer to as the work loop of the application. The work loop of the Read from File thread of the application is shown in Program Segment 3.1.

---

**Program 3.1** Timed Playback Read from File Thread Application Work Loop

---

```

1   Read Frame from File
2   Read Control Variables
3   Process Frame
4   Write State Variables
5   Frame Processing Complete

```

---

Line 1 of Program Segment 3.2 reads a frame from the input video file. Line 2 of Program Segment 3.2 reads the control variables from the SDF that can be used to control the execution behaviour of the application. But the Timed Playback application does not use control variables and so there is not interaction taking place there. Line 3 refers to the processing of a frame by the Read from File thread. The Read from File thread shoves the frame that it read from the input video file to the output queue. Since this is a video pipeline, output queue of this thread will be the input queue of the next stage of the pipeline, the Display thread. Then, the Display thread updates its state variables and writes them to the SDF that governs the TP application. This is shown in Line 4. At the end of processing a single frame, we denote to the SDF that the processing of one frame is complete as shown in Line 5.

The work loop of the Display thread is shown in Program 3.2. This is different from the Read from File thread's workloop. Also, Figure 3.5 shows the various interactions that happen with the SDF during the work loop. It also shows a couple of actions that the SDF takes when some of these interactions take place. The numbers in the figure correspond to the line numbers in the work loop. These

are the descriptions of each of the interactions and the rationale behind each of them.

---

**Program 3.2** Timed Playback Display Thread Application Work Loop

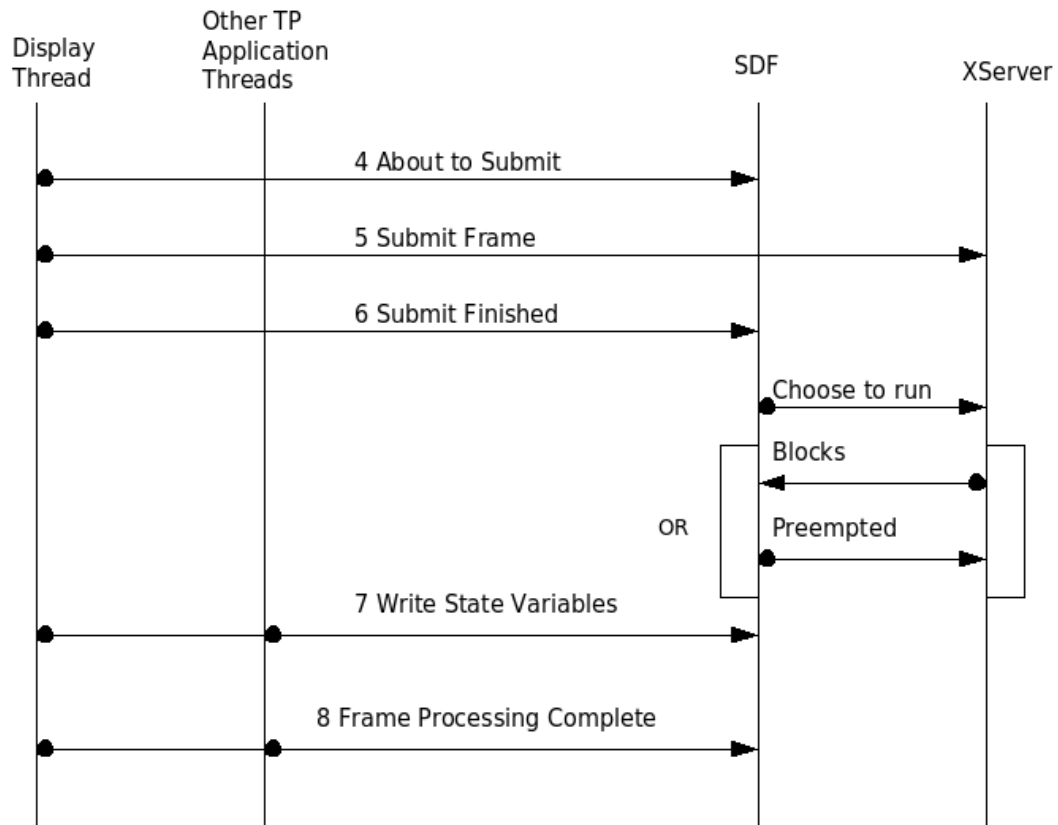
---

```

1  Read Frame from Queue
2  Read Control Variables
3  Process Frame
4      Display Xserver Interaction = About to Submit
5      Send frame to Xserver
6      Display Xserver Interaction = Submit Finished
7  Write State Variables
8  Frame Processing Complete

```

---



**Figure 3.5.** Timed Playback experiment SDF-Application interaction

Line 1 of Program Segment 3.2 reads a frame from the input queue, in the case of the Display thread.

Line 2 of Program Segment 3.2 reads the control variables from the SDF that can be used to control the execution behaviour of the application. But the Timed Playback application does not use control variables and so there is not interaction taking place there.

Line 3 refers to the processing of a frame by the application. When the Display thread is processing a frame it sends the frame over to the Xserver. The SDF has been designed in a way where as soon as the frame is sent to the Xserver, a notification is sent to the SDF. This notification will start the execution interval which will tell the SDF to choose the Xserver thread to run for a pre-determined amount of time. During this whole process, when the call to the Xserver is made from the Display thread, sometimes the Display thread blocks due to the Xserver processing but now always. In order to make sure that the SDF knows which state the application is in, we send a "About to Submit" state to the SDF corresponding to the state variable Display Xserver interaction. We can see this in Line 4 of Program 3.2 and its corresponding interaction with the SDF in Figure 3.5. Once the submission of the frame to the xserver is done, we change the state variable to Submit Finished such that the Xserver execution interval can start. At this point, the SDF decisions are as shown in Figure 3.5 as 'Choose to run'. The 2 conditions when the Xserver is stopped from being chosen is when either the Xserver has blocked which means that it has processed all the frames or when the end of the Xserver execution interval is encountered at which point the Xserver thread is preempted. This state variable is being published only by the Display thread which is being shown in the Figure 3.5 as a dot marked on the line depicting the Display thread.

Line 7 refers to the place where each thread publishes its state variables to

the SDF. This system call is made after the application thread updates the state variables to the latest values.

Line 8 denotes the completion of processing one frame. This is yet another state variable denoting the completion. This information is used by the SDF to make sure that the threads do not greedily use processor time to process all the frames since we need precise processing of frames according to a frame schedule.

The interactions that the various components of the solution made and how the application and the SDF interacted was seen in the above paragraphs. With all this information, the scheduling algorithm of the SDF will be described along with the pseudo-code of the algorithm.

### **Scheduling Algorithm for Timed Playback**

The high level policy for designing a scheduling algorithm for Timed Playback of Video application is to make sure that frames display timing adheres to the frame schedule produced by the user. So, the "Read from File" thread must keep the output queue filled up with at least one frame such that the Display thread is not devoid of input frames when it runs. An execution interval is used to tell the SDF to choose the Display thread at pre-calculated times such that the desired frame schedule is followed. Another execution interval is used to instruct the SDF to choose the Xserver thread for execution after the Display thread sends a frame to the Xserver. The Display thread's execution interval period is pre-determined and is the amount of time it has to process one frame by the Display thread.

The usage of execution intervals to control the timing of execution of threads is called Explicit Plan semantics. This is used for the Display thread and the Xserver thread. For the "Read from File" thread, the policy is to keep the queue filled up

to a pre-determined high queue watermark such that the Display thread is never in want of frames. So, we call this Fill Queue semantics. The Display thread and Xserver interaction that was shown in the interaction diagram in Figure 3.5 is referred as the Xserver runnability semantics. In the Algorithm 3.3, we use the "Drain" semantics as a method used inside the display execution interval. In a video pipeline, "Drain" semantics enforce the condition that the pipeline stage that is farthest downstream with a frame in its input queue must be chosen to run next so that it can drain its input and thus achieve draining the input in the pipeline. The reason for this choice is such that this algorithm can be extended to a Transformation pipeline experiment where there are 1 or more threads in the pipeline in between the "Read from File" and Display threads. The Transformation Pipeline experiment is discussed in detail in Section 4.1.2. This experiment has similar semantics and requirements as the Timed Playback application.

There are 2 phases to the Explicit Plan Video SDF. They are the Bootstrap phase and the Continuous phase. The Bootstrap phase makes sure that the Explicit Plan semantics do not start until the "Read from File" thread has read at least  $N$  frames and sent it to its output queue. The number of frames that the Bootstrap phase allows the "Read from File" thread to read is controlled by a state variable "bootstrap queue length" that the user specifies while setting up the GS hierarchy. This is shown in the Scheduling Algorithm 3.3 from Lines 1 to 4.

The Continuous phase of the SDF is the entire algorithm but for the Bootstrap phase. In this phase, we start by checking whether the trigger to start the Explicit Plan has occurred. If the trigger has not occurred, we do not select any thread as shown in Line 7 of Algorithm 3.3. Once the trigger has been made, we move on

to the main decision making of the SDF from Line 11 of the algorithm. Here, we check if the Display execution interval is active. If not, then the "Fill Queue" semantics are invoked and the "Read from File" thread is chosen until it has filled its output queue upto the "High Queue Watermark". The "Fill Queue" Semantics are shown as a separate function from Line 16 to 20 of the Algorithm.

If the Display execution interval is active, then we check if the number of frames produced by the Display thread is less than the number of display execution intervals that have taken place. This check is made so that the Display thread does not process more than one frame in a single execution interval so that the frame schedule will be maintained. If the state variable indicate that the Display thread has fallen behind schedule, then the Drain semantics which chooses the thread that has processed the least number of frames. If some threads are equal then the one farthest upstream the pipeline would be chosen since the latter would not have frames to process. This way we drain a pipeline. For the Timed Playback application, the Display thread is the only one using Drain semantics which is shown from Line 22 to 28 of Algorithm 3.3.

The Final runnability check for the Display thread and Xserver thread happen from Line 31 to 44 of the Algorithm. Here, depending on the value of the Display Xserver Interaction (DI) state variable, the choice between the Display and the Xserver thread is made. When there is no interaction between Display and Xserver, the Display thread is chosen. When DI has a value of 'About to Submit', which means that the Display thread is about to call the Xserver to display the frame, two possibilities could occur. One, the SDF could see that the Display thread is blocked because the Xserver call made the Display thread to block. If this is case, then the SDF has to choose the Xserver such that it finishes



the processing as fast as possible and returns control back to the Display thread. If the Display thread is runnable, then the Xserver did not make the Display thread block and thus the Display thread is chosen from the SDF. This is shown in the algorithm from the Lines 34 to 37. When the DI value is 'Submit Finished', which means that the frame has been sent to the Xserver and the Xserver execution interval has been activated, we choose the Xserver thread if it is runnable. When the Xserver thread is blocking, we make the assumption that the Xserver has processed all the frames in its queue. So, in this case we stop the Xserver execution interval, set the DI value to 'No Interaction' and choose the Display thread to run. This way the Explicit Plan Video SDF algorithm achieves all its goals.

---

**Program 3.3** Explicit Plan Video SDF Algorithm

---

```
1 Has the application bootstrapped
2   ->No : If (read_mpeg_output_queue_length < bootstrap_queue_length)
3           Choose Read from File Thread
4   ->Yes: Goto Step 1
5 Step 1:
6 Has the frame schedule started
7   ->No: Do not select any thread. Wait for the schedule to start
8   ->Yes: Goto Step2
9
10 Step2:
11 Is an Display execution interval active
12   -> No - Fill Queue Semantics()
13   -> Yes - If (display_thread_frames_produced < no_of_display_exec_intervals)
14           Drain Semantics()
15
16 Fill Queue Semantics()
17   If (read_mpeg_output_queue_length < high_queue_watermark):
18       Choose Read from File Thread
19   Else:
20       Return nothing from SDF
21
22 Drain Semantics()
23 Does it have input frames in the queue
24   -> No - Choose Read from File Thread
25 Compare frame progress
26   -> Choose the node with the least number of frames
27   -> If equal, choose the node that is farthest upstream in the pipeline
28       Goto step 3
29
30 Step 3:
31   If (Chosen Node == Display Thread):
32       If (DI == No Interaction)
33           Choose Display Thread
34       Else if (DI == About to Submit and DT is runnable)
35           Choose Display Thread
36       Else if (DI == About to submit and DT is blocked)
37           Choose Xserver
38       Else if (DI == Submit Finished)
39           If (Xserver execution interval is active):
40               If Xserver is runnable : Choose Xserver
41               If Xserver is blocked:
42                   Stop Xserver execution interval
43                   Set DI = No Interaction
44                   Choose Display Thread
```

# Chapter 4

## Experiment Design

This chapter explains the overall experimental design used to demonstrate how State and Control Variable Programming Model(SCPM) can be used to improve performance of applications requiring them. In this work we used video based applications requiring precise frame display behavior as our driving examples. These are discussed in Section 4.1

A prominent part of any customized programming model is the scheduling hierarchy used to control application behaviour. In our driving examples we have used 3 different scheduling semantics: We have the Self Timed (ST), Read and Display(RD) and Read,Display and Xserver (RDX). The first represents the current practice in writing these types of applications and demonstrates why customized programming models are required. The second, RD, places the application threads under Group scheduling control and shows significant improvement. The third, RDX, shows that placing the Linux Xserver thread under Group Scheduling control provides further improvement and demonstrates the general principle that many linux system level computations play an important role in application performance and any customized programming model framework which intends to

provide precise computation control must include the ability to control Linux system service computations in the programming model. These 3 scheduling semantics are discussed in more detail in Section 4.2.

A fundamental aspect of the design of any experiment is the selection of performance metrics. We have selected 2 metrics which are different views of the same fundamental behaviour. Since frame display timing precision is our main concern we have chosen metrics that consider the timing accuracy of frame display. We define Frame Display Timing Error (FDTE) as the absolute value of the difference between when a frame was scheduled for display and when it was actually displayed. Unfortunately, there is no way for us to precisely measure when the pixels of a frame are displayed on the screen. We are thus forced to approximate the ideal metric in 2 ways. The first is a metric telling us when a frame is submitted to the Xserver which we name FDTE-Submission (FDTE-S). The second is a metric telling us when the first execution interval of the Xserver thread completed after frame submission occurs. This is what we call FDTE-Completion (FDTE-C). These metrics and how they are measured are discussed in greater detail in Section 4.3

## 4.1 Applications

Our example applications that demonstrate the SCPM uses the KU Image processing library which has been discussed in 3.1.4.

Video applications require the video to be displayed on the screen in a timely fashion so that the user perceives it in the proper manner. Under standard Linux, the available programming model (usage of sleep system calls with priority scheduling) doesn't enable the developer to control when frames are displayed

precisely enough under all system conditions. Developers of video also need to be able to see the effects of video processing anomalies caused by unreliable timing of frame delivery. Viewed another way, experimenting with ways to adapt to varied frame delivery times requires the ability to precisely control when the frames are delivered to the application.

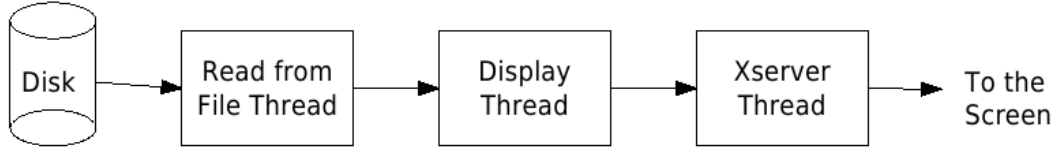
The Timed Playback application is thus an excellent way to demonstrate the effects of various frame delay and jitter values by being able to precisely follow specified schedules which represent various scenarios for frame delay and jitter.

The Transformation Pipeline goes beyond the simple Timed Playback application to show that arbitrary computation structures can be represented using the SCPM, particularly the Group Scheduling hierarchy structure, to enforce the desired execution behaviour.

For these types of applications, a programming model that is able to specify the semantics of the application to the scheduling system will be able to demonstrate better control of the application than one that does not, such as that of standard Linux.

#### **4.1.1 Timed Playback**

This application controls playback of video according to a pre-determined frame schedule. The frame schedule is a specification of when each frame in a given video should be displayed on the screen. The specification of various schedules is useful in demonstrating to users the effects of various frame delay and jitter values. In addition, a frame delivery computation or pipe segment which is following a specific schedule can be valuable for reproducibly testing behaviour video processing pipelines under specific frame delivery delay and jitter values.



**Figure 4.1.** Timed Playback Application Pipeline Structure

This application consists of 2 threads in the KUIM domain and the Linux Xserver thread as illustrated in Figure 4.1. One of them is the "Read from File" thread and the other is the Display thread that sends the frame over to the Linux Xserver. The Xserver thread controls actual frame display and thus has a fundamental influence on performance under our chosen metric of FDTE.

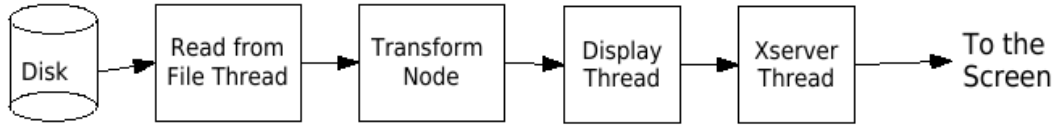
#### 4.1.2 Transformation Pipeline

The Timed Playback application discussed in Section 4.1.1, while important in its own right as a precise demonstrative environment for video behaviour, can also be viewed as the simplest possible video computation structure. To demonstrate that our approach to customized programming models can easily represent arbitrary computation structures, we extend the Timed Playback architecture with 1 transformation node with arbitrary character.

This pipeline application is an extension of the Timed Playback application as it adds 1 video transformation node in between the read mpeg from file thread and the Display thread.

## 4.2 Schedulers

Applications like Timed Playback and Transformation Pipeline require precise and application specific computation control which is enforced by the scheduling



**Figure 4.2.** Transformation Pipeline Application Pipeline Structure

algorithm that is used in the system. For the applications explained in the preceding section, there are 3 different scheduling algorithms that have been used. These schedulers implement the semantics that is required for the applications.

The Self Timed is a basic scheduler that uses nanosleep system calls from the application to implement the semantics required for the application. This scheduler uses the standard Linux scheduler based programming model.

The Read and Display (RD) scheduler uses GS to implement the semantics of the application. This scheduler controls the "Read from File" thread and the Display thread but does not take the Xserver thread into account.

The Read, Display and Xserver (RDX) scheduler is a variant of the RD scheduler. This scheduler brings the Xserver thread under control in addition to the "Read from File" and Display thread so as to better satisfy the application requirements.

### **Self Timed with no Group Scheduling (ST)**

Self timing exerts an indirect influence on when the Display thread runs through manipulation of when the thread is blocked. But the Standard Linux scheduler still decides when to run the thread using its own semantics. The key weakness in this approach is that just because the thread is ready does not mean that it will run as there are many factors in the system scheduling decision that

determine when a ready process will actually run. The standard Linux programming model provides developers with only process priority and process state manipulation through `nanosleep` and other system calls to influence the scheduling decision.

The work loop of the application implementing the ST semantics is shown in Program Segment 4.1. The work loop reads a frame from the input queue and sends the frame over to the Xserver. Self timing is achieved by the use of the `nanosleep` system call. The amount of time for the process to block is calculated in a separate function as shown in Line 6 of Program 4.1. This function, whose algorithm is shown in Program 4.2, uses the frame schedule that is specified for the application and uses the frame delay values in the schedule to calculate the time to sleep. It gets the current time using `gettimeofday()` function and calculates the `nanosleep` time from the absolute time at which the next video frame should be displayed. This value is then sent as the argument to the `nanosleep` system call in Line 7 of Program 4.1.

---

**Program 4.1** Self Timed Code Structure

---

```

1  while (input_queue != EMPTY) {
2
3      current_frame = input_queue.RemoveFrame();
4
5      send_current_frame_to_xserver();
6      nanosleep_time = get_next_nanosleep_time(frame_schedule);
7      nanosleep(nanosleep_time);
8  }
```

---

## Read and Display Control(RD)

The "Read from File" thread and the Display threads are put under exclusive Group Scheduling control. The group hierarchy places both these threads under



---

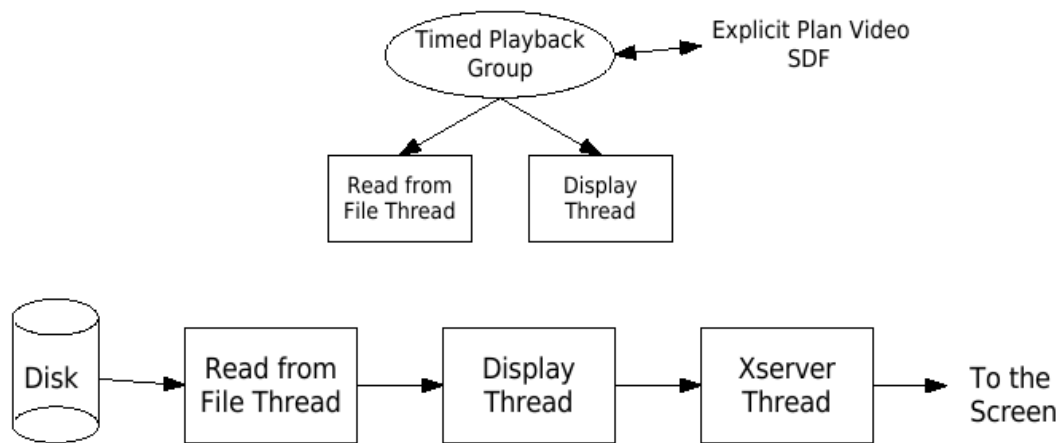
**Program 4.2** Self Timed Nanosleep time calculation function

---

```
1  get_next_nanosleep_time(frame_schedule) {  
2  
3      current_time = gettimeofday();  
4      next_video_display_time = current_time + next_frame_schedule_interval;  
5      nanosleep_time = next_video_display_time - current_time - time_of_work_loop;  
6  
7      return nanosleep_time;  
8  }
```

---

one group as illustrated in the Figure 4.3, the scheduling of group members is governed by the Explicit Plan Video SDF that was discussed in Section 3.3.



**Figure 4.3.** Timed Playback Group Scheduling Hierarchy with RD Scheduler

The Explicit Plan Video SDF makes sure that the "Read from File" thread keeps the output queue (the Display thread's input queue) non empty by providing it processing time during non execution interval time, which is determined by the frame schedule of the Display thread. Then, during the execution interval of the Display thread, the SDF affirmatively decides to run the Display thread depending on its state being runnable. The Display thread then submits the frame to the Xserver for display but since the Xserver is not under GS control, the Standard

Linux scheduler determines when the Xserver runs.

The work loop of the Display thread under the RD scheduler starts with it reading a frame from its input queue and sending it to the Xserver as in Line 5 of the Program 4.3. It then writes the state variables of the Display thread to the SDF using a system call to the GS kernel API. Once the processing of one frame is done it sends a notification of completion of processing of a frame to the SDF as shown in Line 7 of the Program segment.

---

**Program 4.3** Read and Display Code Structure

---

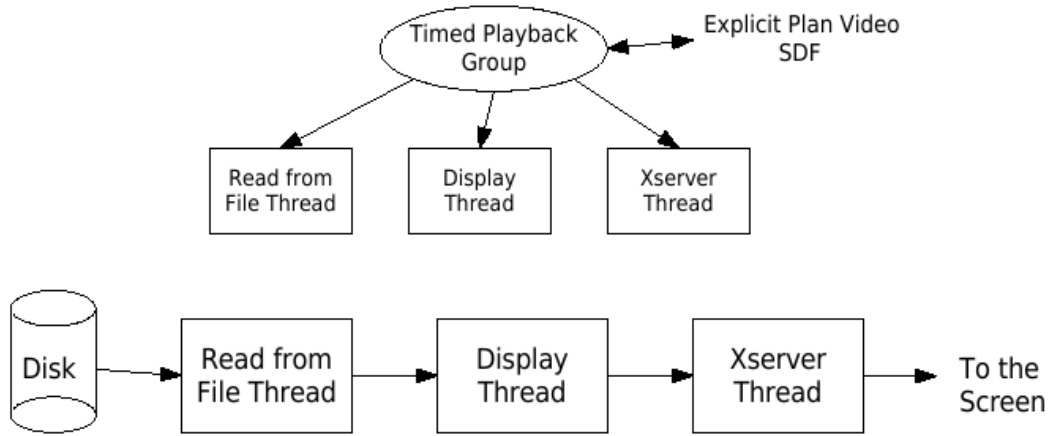
```
1  while (input_queue != EMPTY) {  
2  
3      current_frame = input_queue.RemoveFrame();  
4  
5      send_current_frame_to_xserver();  
6      write_state_variables_in_groupsched();  
7      frame_completion_notification_to_groupsched();  
8  }
```

---

### Read, Display and Xserver Control(RDX)

The scheduling semantics of this SDF adds the Linux Xserver to the computation group to exert as much control as possible over when the frame is actually displayed. However, since we did not want to have to modify the Xserver code we are limited to giving the Xserver an opportunity to execute until it blocks as our best approximation of controlling when a frame is displayed. We are making an assumption, which we believe is reasonable, that if we let the Xserver run until it blocks after the Display thread has submitted a frame then the submitted frame will have been displayed. The GS hierarchy of the application under the RDX scheduler is shown in Figure 4.4. This shows that the Read from File, the Display and the Xserver threads are under the Timed Playback group and they

are governed by the Explicit Plan Video SDF.



**Figure 4.4.** Timed Playback Group Scheduling Hierarchy under RDX Scheduler

FDTE-S and FDTE-C are bounding metrics on FDTE which is discussed in Section 4.3. The FDTE-S variant occurs before the frame is displayed even though we do not know by how much. The FDTE-C variant is calculated from the first time the Xserver blocks after it was provided a chance to run by the SDF which is a worst case frame display time by our assumptions. The Read from file thread, the Display thread and the Xserver thread are put under exclusive GS control. The group hierarchy places all these three threads are under one group, the scheduling of which is governed by the Explicit Video SDF that was introduced in the Section 3.3 and shown in Program 3.3 in Page 46.

The Display thread notifies the SDF of the frame about to be sent to the Xserver. Once the SDF has been notified, it decides to schedule the Xserver for a pre-determined amount of time until the Xserver blocks. The pre-determined amount of time has been calculated by finding out the amount of time that the Xserver thread runs in the system during the processing of a frame by the Display

thread. The Xserver blocking is considered as an indication that the Xserver has completed displaying all the frames in its queue which means that the frame sent from the Display thread has been displayed as well. But, this is only an approximate measure since we are not absolutely sure that the Xserver has displayed our frame as measuring that would require instrumenting the Xserver code which is beyond the reasonable scope of this work. Note however that anyone who wanted more precise control could use the SCPM within the Xserver code in order to place frame display under more precise control. However, simply controlling when the Xserver thread executes by placing it under GS control improves the visual quality of the video. This is however only a qualitative measure.

The work loop of the Display thread under the RDX scheduler is similar to that of the Display thread under the RD scheduler but for one difference in its interaction with the SDF with respect to the Xserver. From Program Segment 4.4 we can see that at Line 6 we send a notification to the SDF asking it to start the Xserver execution interval. This call is being made as soon as the Display thread sends the frame over to the Xserver. This way the SDF will be able to control the execution of the Xserver thread and thus exhibit better control.

---

**Program 4.4** Read, Display and Xserver Code Structure

---

```

1  while (input_queue != EMPTY) {
2
3      current_frame = input_queue.RemoveFrame();
4      read_control_variables_from_groupsched();
5      send_current_frame_to_xserver();
6      start_xserver_execinterval_notification_to_groupsched();
7      write_state_variables_in_groupsched();
8      frame_completion_notification_to_groupsched();
9  }
```

---

## 4.3 Metrics

In the experiments that have been discussed, timely frame display on the screen is considered to be the relevant performance metric since only when the display is timely, the user perceives the video properly. All the components of the application are designed in order to make frame display more accurate and precise. But, as with any system there is a difference between when the user wants a frame displayed and when the frame is actually displayed on the screen. The main metric of this experiment is this difference. We call this metric Frame Display Timing Error (FDTE). But, this is an ideal metric as the exact time at which a frame gets displayed on the screen depends on when the Linux Xserver sends it to the display driver and when the driver sends it to the monitor. So, as an approximation of this metric we measure the FDTE-Submission and the FDTE-Completion times.

### **Frame Display Timing Error - Submission Metric**

The FDTE-S is the difference between the time that a frame was sent to the Xserver for display to the time that it was supposed to be displayed according to the frame schedule. We call this FDTE-S because the time measured here is not when the frame was displayed on the screen but is when the frame was sent to the Xserver by the Display thread.

### **Frame Display Timing Error - Completion Metric**

The FDTE-C is the difference between our approximation of the time the Xserver displayed the frame to the time that it was supposed to be displayed according to the frame schedule. Here our approximation considers the Xserver

frame completion time to be the time at which the system context switched away from the Xserver thread for the first time after the frame was submitted to the Xserver by the Display thread.

These two metrics are measured from Datastream events. The events are shown in the Program Segment 4.5. The FDTE-S in Line 5 is an event just before the frame is sent to the Xserver. The FDTE-C in Line 8 is an inference that is made by merging the event from the Display thread with the context switch data of the Xserver thread from the Kernel at the time when the work loop was in progress. We define the FDTE-C metric as the first time the Xserver thread blocked after the Xserver execution event notification is sent to the SDF.

---

**Program 4.5** FDTE Metric measuring Datastream events in the work loop

---

```

1  while (input_queue != EMPTY) {
2
3      current_frame = input_queue.RemoveFrame();
4      read_control_variables_from_groupsched();
5      FDTE-S_DATASTREAM_EVENT();
6      send_current_frame_to_xserver();
7      start_xserver_execinterval_notification_to_groupsched();
8      FDTE-C_DATASTREAM_EVENT();
9      write_state_variables_in_groupsched();
10     frame_completion_notification_to_groupsched();
11 }

```

---

## 4.4 Experimental Conditions

Our goal is to create programming models and system support under which applications will execute according to their stated requirements and constraints under all possible conditions. The SCPM permits developers to create customized SDF's as part of ensuring that developers can create programming models satisfying application specific semantics. The general computation load on the system

while our driving applications are executing clearly influences their behaviour and thus influences whether they execute according to their requirements and constraints.

Note that any scheduling related problem is usually either non existent or trivial under lightly loaded conditions since, at the logical extreme, if only one process is available all schedulers will make the same decision. In our experimental design we have chosen 2 loading conditions under which we test the SCPM to show it produces better performance than the baseline ST programming model and standard Linux scheduler.

One of the experimental conditions we have chosen is "Lightly Loaded" where our applications execute on a system with all other normally present Linux processes but no other significant application is running. The other experimental condition we chose is "Highly Loaded" which adds significant CPU, interrupt and network load to the Lightly Loaded conditions. The CPU load is generated using 2 continuously repeating kernel compilations. The disk interrupt load is generated by one of the kernel compiles which uses the local disk. The network interrupt and bandwidth load is created by one of the kernel compiles which accesses a remotely mounted NFS drive and by a secure copy (SCP) process which is a greedy copy process transferring a remote file system image onto the local disk and the local disk write end of the SCP.

The main purpose of the Lightly Loaded condition is to show any advantages for application behaviour that are inherent to the application aware SDF semantics compared to the Standard Linux scheduler which controls each thread individually and is unaware of the relationships among threads created by the application structure. The main purpose of the Highly Loaded condition is to see how robust

the application aware SDF's are in achieving the desired behaviour when a large amount of competition for resources at both the user level and the OS level are present.



# Chapter 5

## Experimental Results

Chapter 4 described the driving applications we used as the experimental framework for the results presented here. The Timed Playback experiment, explained in Section 5.1, and Transformation pipeline experiment, explained in Section 5.2, represent video processing experiments with application semantics based scheduling constraints which serve as good examples of the basic problem of controlling applications according to customized semantics as described in Chapter 1.

Chapter 4 also described the Self Timed(ST), Read and Display (RD) and Read, Display and Xserver (RDX) schedulers used to control these applications in various experiments as well as the FDTE-S and FDTE-C performance metrics used to evaluate the performance of the applications under each scheduler's control.

Recall, as discussed in Section 4.3, that the ideal metric is the absolute difference between the actual time at which the pixels of a frame are displayed on the screen to the user and the time at which that display was scheduled. However, since we are unable to capture a timestamped event when the pixels are displayed we must approximate the ideal FDTE metric instead. FDTE-S is essentially a

best case approximation of the unmeasurable ideal FDTE and the FDTE-C is a worst case approximation of the ideal metric.

The rest of this chapter presents our experimental results which demonstrate that providing a Customized SDF as part of support for the Customized application Programming Model improves performance under both best case FDTE-S and the worst case FDTE-C approximations of the ideal FDTE metric. Section 5.1 discusses how the Timed Playback application performs under both lightly and heavily loaded conditions using the 3 SDF's. Section 5.2 discusses how the transformation pipeline application performs under both lightly and heavily loaded conditions using the ST, RD and RDX schedulers. Section 4.4 discussed the details of the Lightly Loaded and Heavily Loaded conditions used for the experiments. Note however that the RD and RDX schedulers were modified slightly to appropriately control the Transformation computation threads added to the Timed Playback application. Section 4.1.2 explained how the modified RD and RDX algorithms differ from the versions used for the Timed Playback applications.

## 5.1 Timed Playback

This section discusses the experimental results of the Timed Playback application under both Lightly Loaded and Highly Loaded experimental conditions. Performance is measured using the FDTE-S and FDTE-C metrics and we compare the performance under the 3 standard scheduling semantics of ST, RD, RDX.

During the discussion of the results of the Timed Playback application, certain notations are used in order to describe a particular experiment. The notation follows the convention of "metric-scheduler type-load type". Here, the metrics in this experiment are FDTE-C and FDTE-S which are represented as FDTEC and

FDTES. The 3 scheduler types are referred to as ST, RD and RDX. The load type refers to the kind of experimental conditions this application was run under. They are high load, referred as HL, and light load, referred as UL. So, the notation of the FDTE-C results when the application was run under high load with the RD scheduler would be FDTEC-RD-HL.

### **5.1.1 Lightly Loaded Conditions**

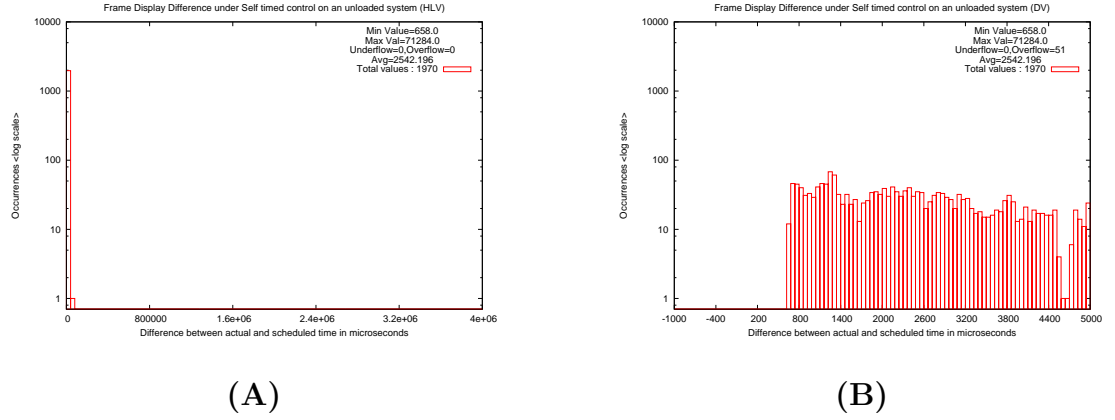
Here we examine the results under the Lightly Loaded condition where the application runs on a system that is otherwise idle and thus contains only the normal background, system service and desktop application processes.

#### **5.1.1.1 Frame Display Timing Error -Submission (FDTE-S)**

The metric used in these set of results are FDTE-S which is a best case approximation of the ideal FDTE metric. We examine the results for all the 3 types of scheduling semantics under this metric.

#### **Self Timed Control(ST)**

Figure 5.1(A) shows a histogram of the entire data set. This shows that the vast majority of the FDTE-S values are clustered in the lowest bucket. Normally, it would be pointless to display such a graph since it contains almost no information. However, we include it here because comparing it to a similar graph for behaviour under highly loaded conditions will be important. Figure 5.1(B) shows an expanded view of this data set which shows that of 1970 FDTE-S values all but 51 values fell below 5000 microseconds(5 milliseconds). The minimum value was 650 microseconds and the roughly 1900 sample points are uniformly distributed



**Figure 5.1.** FDTE-S under Self Timed with Light Load (FDTE-ST-UL) (A)High Level View(HLV) (B) Detailed View(DV)

between the minimum and 5000 microseconds. In Figure 5.1(A) all but one of the samples overflowing Figure 5.1(B) still fall in the first bucket. Figure 5.1(A) is on such a large scale because Self Timed's performance heavily degrades under Heavily Loaded conditions and we wanted this histogram and the corresponding histogram on the same scale.

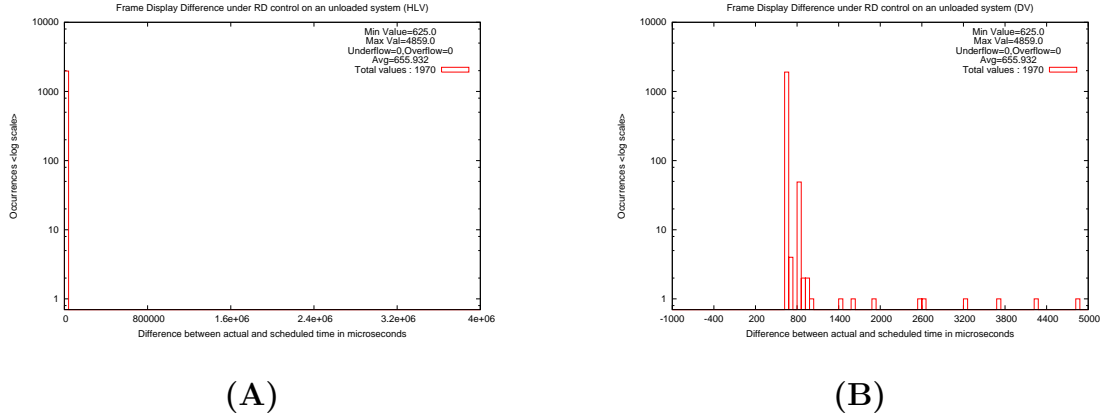
Typical video has a frame rate between 24 and 40 fps making the execution slot for each frame range from 40 and 25 milliseconds. In general terms, single frame errors and delays are not perceived by a human viewer. Figure 5.1 (A) shows that the single frame in the second bucket had a delay of 70 milliseconds and all other buckets had a delay of 40 milliseconds indicating that a human viewer might have seen atmost one glitch under these conditions.

The average FDTE-S delay value of this data set was 2.5 milliseconds which is below the human perception threshold. Thus the application meets its requirements and constraints under the ST model when the system is lightly loaded. However this is unsurprising since all scheduling algorithms act the same way

when there are few if any choices to be made.

Even though the range of frame submission error from 650 to 5000 microseconds under ST is under human perception threshold, for video applications it is still a relatively broad range which would not be appropriate for other classes of applications such as data acquisition or machine control. Later sections will show that the RD and RDX schedulers produce a tighter distribution and will thus be more appropriate for applications with more stringent requirements or under less forgiving conditions.

### Read and Display SDF Control(RD)



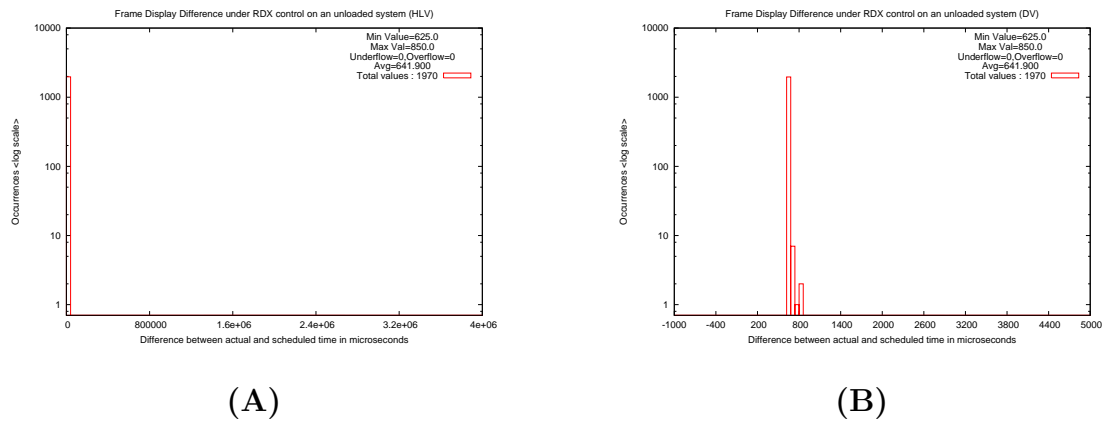
**Figure 5.2.** FDTE-S under RD Control with Light Load (FDTES-RD-UL) (A)High Level View (HLV) (B) Detailed View(DV)

Figure 5.2(A) shows a histogram of the entire data set. This shows that all the FDTE-S values are clustered in the lowest range. Although this graph does not show a lot of information, it will be useful when we look at the performance of the ST scheduler with heavy load.

Figure 5.2(B) shows an expanded view of the data set. This shows that all

the 1970 FDTE-S values fall below 5000 microseconds. More than 95 percent of the values are distributed between the minimum value of 625 microseconds and 680 microseconds which provides a considerably tighter distribution than that for ST as shown in Figure 5.1(B). Though it provides tighter distribution, the human cannot appreciate the improved performance over that of ST shown in Figure 5.1 since the FDTE-S values are below that of human perception. Also, the values here are not of great significance because they are being retrieved from a system running under light load which makes TP performance under control of the standard Linux scheduler to be as good as that under GS due to the reduced contention for the processor. That the values are not significant is evident from the Figure 5.1. Note that there are about 10 values that fall between 1000 microseconds and 5000 microseconds but these are still too far under human perception to be worried about.

### Read, Display and Xserver threads control(RDX)



**Figure 5.3.** FDTE-S under RDX Control with Light Load (FDTE-S-RDX-UL) (A)High Level View(HLV) (B) Detailed View(DV)

Figure 5.3(A) is a histogram of the FDTE-S values when the system is under control of the RDX scheduler with Light Load on the system. This histogram gives a view of the values in the whole data range. Also, this figure is identical to Figure 5.2(A) as the FDTE-S metric is not affected by adding the Xserver thread to the GS hierarchy.

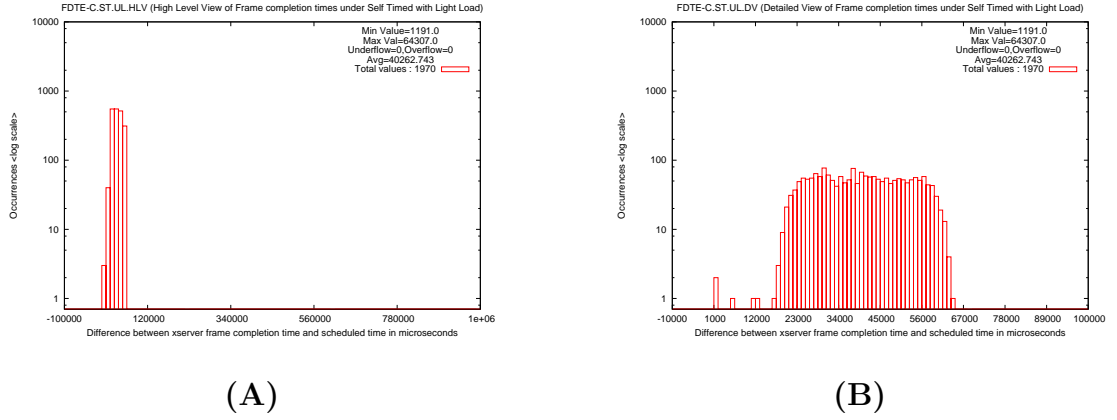
Figure 5.3(B) shows an expanded view of the data set. This shows that all the FDTE-S values fall in the range between 625 and 850 microseconds. This graph also looks very similar to the Figure 5.2(B) as the FDTE-S metric is calculated from the event that happened just before the frame submission to the Xserver took place. This means that the aspect of RDX that is different from RD is not going to affect the outcome of the results measured with this metric. This graph provides a considerably tighter distribution than that for ST as shown in Figure 5.1(B). Also, the values here are not of great significance because they are being retrieved from a system running under light load which permits the Standard Linux scheduler to perform as well as GS due to the low level of contention for the processor.

#### **5.1.1.2 Frame Display Timing Error - Completion (FDTE-C)**

The metric used in these results are FDTE-C which is a worst case approximation of the ideal FDTE metric. We examine these results for the 3 types of scheduling semantics under this metric.

#### **Self Timed Control (ST)**

Figure 5.4(A) shows a histogram of the entire data set. This shows that all the values are clustered towards the lower range of values. This graph does not contain



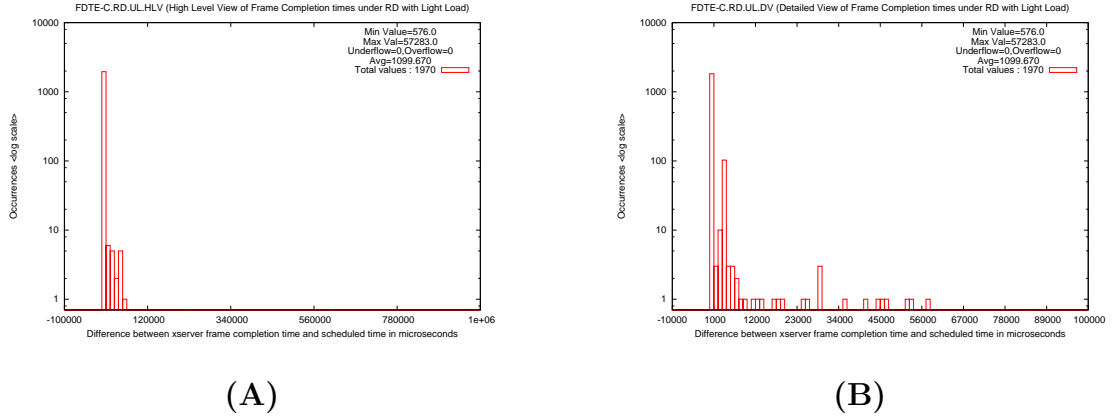
**Figure 5.4.** FDTE-C under Self Timed with Light Load (FDTEC-ST-UL) (A)High Level View(HLV) (B) Detailed View(DV)

a lot of information since it was gathered under a system with light load but it is important to notice that the worst case FDTE-C average is 40 milliseconds in this experiment. This metric takes into account the execution behaviour of the Xserver thread due to which the FDTE-C average value is substantially larger than that of the FDTES-ST-UL. We would see that under heavily loaded conditions, the performance measured by this metric gets worse.

When looking at the graph in a detailed manner, as provided by Figure 5.4(B), we can see that the values are uniformly distributed between 18000 microseconds and 62000 microseconds. The delay in frame display in this experiment can be as much as 62000 microseconds which means that a frame has been displayed on the screen when it is past the scheduled time of display for the next frame by 50 percent for a 40 ms frame period. This indicates relatively poor performance even under a lightly loaded system and this forms one of the major problems that is being addressed the SCPM. You will see in the graphs discussed next that the performance of our application aware SCPM is extremely good.



## Read and Display Control (RD)



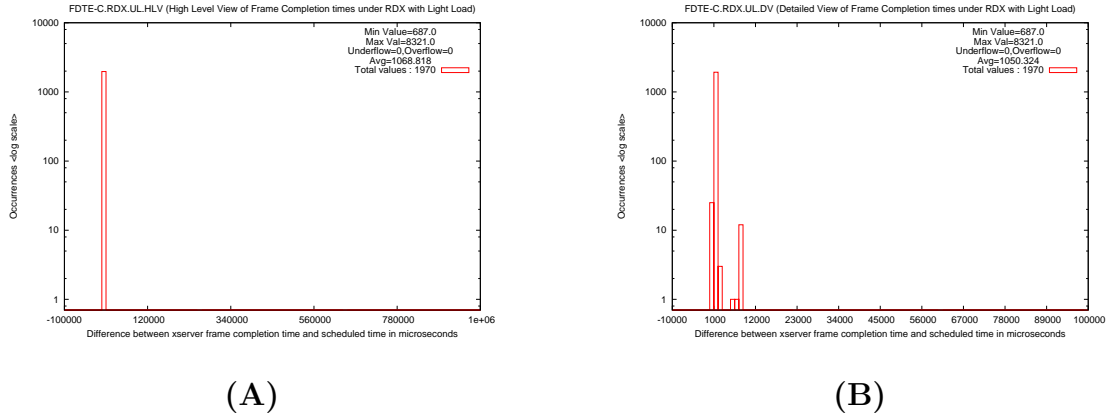
**Figure 5.5.** FDTE-C under RD Control with Light Load (FDTEC-RD-UL) (A)High Level View(HLV) (B) Detailed View(DV)

Figure 5.5(A) shows a view of the entire data range and this graph shows us that almost 90 percent of the values are clustered in the first positive bucket and the values range from 576 to 11000 microseconds. The rest of the values are outliers to the first bucket and they are a very small number of values that are not big enough for the user to perceive. The reason for a good performance under this metric is that there are not many competing processes vying for the processor when under light load due to which the Xserver thread is being scheduled in a timely fashion.

Figure 5.5(B) shows the detailed view of the graph and from it we can see that there are a very few frame delay values that are above 5000 microseconds which means that the performance of RD under Light Load is pretty good and even the worst case metric shows good performance. This also shows that Read and Display under light Load is much better than Self Timed when we compare this graph with that of Figure 5.4(B).

Meanwhile, we can also see the effects of the Xserver thread not being under Group Scheduling control as we have outliers in the graph that are as big as 57283 microseconds (57 milliseconds) which can be perceived by the end user. This can be prevented by exhibiting Group Scheduling control of the Linux Xserver thread whose results are presented next.

### Read, Display and Xserver Control(RDX)



**Figure 5.6.** FDTE-C under RDX Control with Light Load (FDTEC-RDX-UL) (A)High Level View(HLV) (B) Detailed View(DV)

Figure 5.6(A) shows a high level view of the data range. You can see that all the values are clustered towards the lowest positive range of the histogram. Under light load, this is only a slight increase in performance when compared to the FDTEC-RD-UL in Figure 5.5 which had a few outliers. This is due to the fact that there is very little competing load which makes the Standard Linux scheduler almost as well as the customized RDX scheduler when deciding to schedule the Xserver thread.

Figure 5.6(B) shows a detailed view from which we can see that more than 99

percent of the delay values are between 687 microseconds and 2100 microseconds which is very tight distribution compared to Figure 5.4(B) and Figure 5.5(B) and shows that there is some advantage of having the Xserver thread under GS control. Also, this advantage will be even more pronounced when we compare the results of RDX with RD and ST when the system is under heavy load. Also, this graph adds to the argument that even with the worst case approximation provided by FDTE-C, the SCPM we have created can ensure better control of the application compared to that of the ST application.

### **5.1.2 Heavily Loaded Conditions**

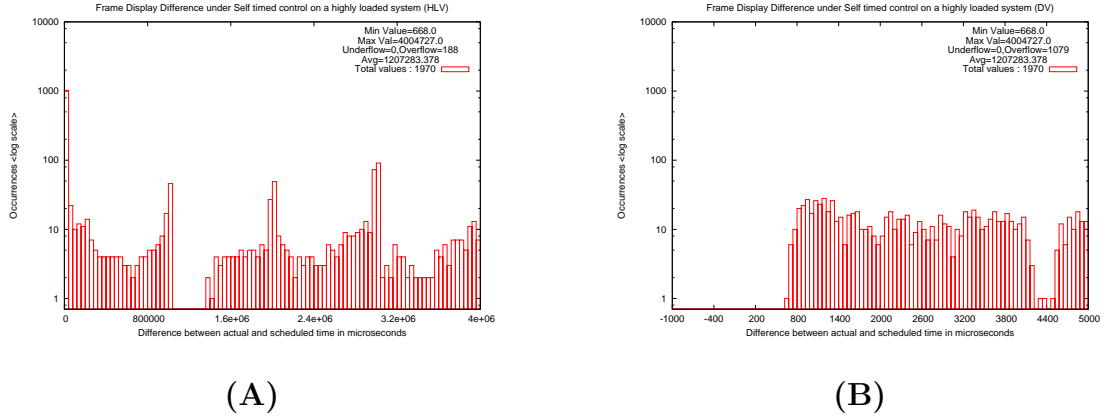
In this section we examine the results of the Timed Playback application running with High Load on the system. The load characteristics on the system are high CPU utilization processes, network interrupts and disk interrupts along with the basic system services. This load was created by 2 continuous Kernel compilations; one using the local disk and the other using a remote NFS file system, and the secure copy (scp) of a remote directory over the network to the local disk.

#### **5.1.2.1 Frame Display Timing Error - Submission (FDTE-S)**

The metric used in these set of results are FDTE-S which is a best case approximation of the ideal FDTE metric. We examine the results for all the 3 types of scheduling semantics under this metric.

#### **Self Timed Control (ST)**

Figure 5.7(A) shows a histogram of the entire data set. This shows that all the FDTE-S values are distributed across the histogram range from 668 microseconds



**Figure 5.7.** FDTE-S under Self Timed with High Load (FDTE-S-ST-HL) (A)High Level View(HLV) (B) Detailed View(DV)

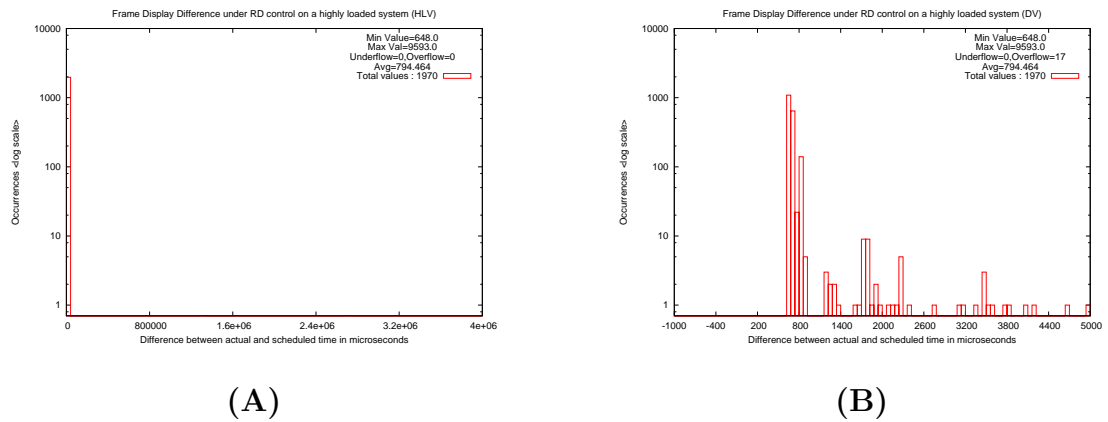
to 4 seconds (which is the highest range of the histogram). This is extremely poor performance as half of the values are greater than 40 milliseconds (40000 microseconds) which means that once every 2 frames there is a delay that would make the video inconsistent with the specified schedule. Also, since this ranges all the way to 4 seconds the user will see random stops in video frame display.

Figure 5.7(B) shows an expanded view of the data set. Notice that more than half of the values are overflowing the high range of 5000 microseconds for this histogram. For the rest of the values, they are uniformly distributed between 668 microseconds and 5000 microseconds as we saw in the Figure 5.1(B). These values are a result of the ST with standard Linux programming model not being able to handle the TP application due to huge competition for the CPU by the many processes running in the system.

That so many histogram overflows have occurred in this experiment supports our argument that in a system that is under stress, the ST with standard Linux programming model does not perform well. When we compare the results of this

experiment with that of RD under high load in Figure 5.8 and RDX under high load in Figure 5.9, it will be clear that the SCPM is better than that of the ST standard Linux programming model.

## Read and Display Thread Control (RD)



**Figure 5.8.** FDTE-S under RD Control with High Load (FDTE-S-RD-HL) (A)High Level View(HLV) (B) Detailed View(DV)

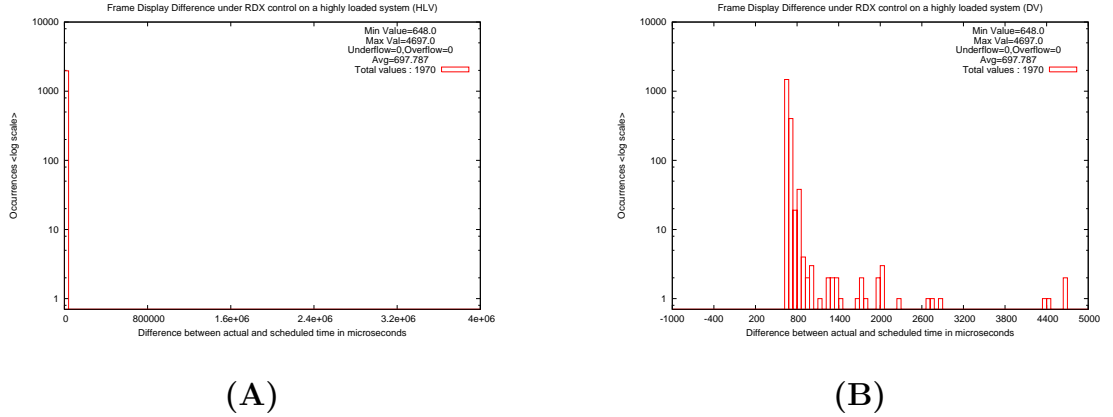
Figure 5.8(A) shows a histogram of the entire data set. This shows that the all the FDTE-S values are clustered in the lowest range. When compared with Figure 5.7(A), it shows that with RD control the performance of the system is significantly better than that under ST control for Highly Loaded conditions. The reason for this is that GS takes precedence over the Linux scheduler and schedules the RD threads without interference from other Linux processes including those producing the high load.

Figure 5.8(B) shows an expanded view of the data set. The distribution of FDTE-S values in this histogram is such that the values of more than 90 percent of the 1970 frames fall between the minimum value of 648 microseconds and 860

microseconds and the other 10 percent indicate that the corresponding frames are displayed with delays between 1000 microseconds and 9593 microseconds. These are delays that are not visible to the human eye.

When comparing these results with those shown in Figure 5.2(B), we can see that the average under light load was 655 microseconds, whereas it is 794 microseconds in Figure 5.8(B). Also, the max value has increased from 4859 microseconds in the Lightly Loaded case to 9593 microseconds in the Highly Loaded case. But this degradation in performance is very small and again it cannot be detected by the user at all. This shows that the SCPM performs well under high load.

### Read, Display and Xserver threads control(RDX)



**Figure 5.9.** FDTE-S under RDX Control with High Load (FDTE-S-RDX-HL) (A)High Level View (HLV) (B) Detailed View (DV)

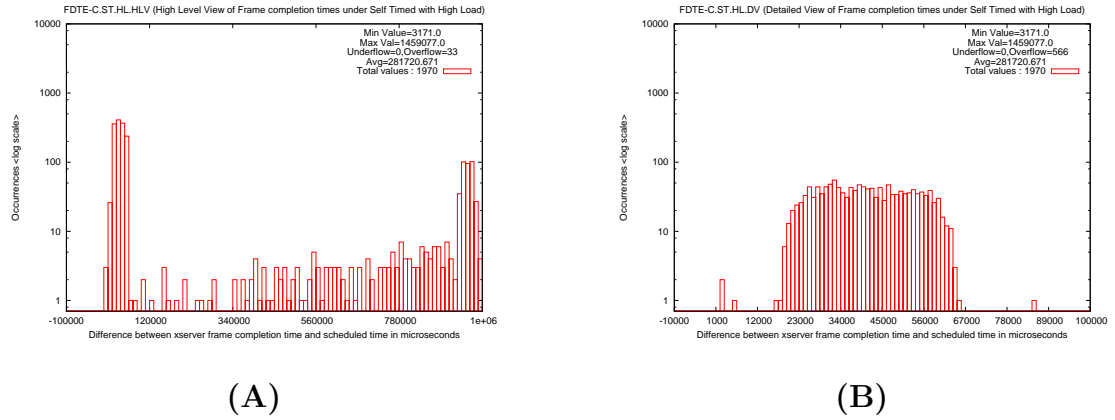
Figure 5.9(A) shows a histogram of the entire data set. Note that all the values are clustered in the lowest bucket. The histogram is identical to that of 5.8(A). It shows that under RDX control the performance of the system is significantly better than the ST control under high load shown in Figure 5.7(A). The reason

being that GS takes precedence over the Linux scheduler and schedules the RDX threads without interferences from other Linux processes.

Figure 5.9(B) shows an expanded view of the data set. This shows that all the FDTE-S values fall below 5000 microseconds. Again, more than 90 percent of the values fall between the minimum value of 648 microseconds and 860 microseconds. There is little or no difference in the values of FDTE-S in this histogram and Figure 5.8(B) as the FDTE-S metric is calculated from the event that happened just before the frame submission to the Xserver and this means that the aspect of RDX that is different from RD is not going to affect the outcome of the results measured with this metric.

#### **5.1.2.2 Frame Display Timing Error - Completion (FDTE-C)**

The metric used in these results are FDTE-C which is a worst case approximation of the ideal FDTE metric. This metric is calculated by computing the difference between the time at which frame submission to the Xserver occurred and the time at which the Xserver context switched out of the CPU for the first time after frame submission. The assumption here is that the reason for the Xserver to context switch out would be because there is nothing more in its queue and that it has blocked. This value will always be larger than the FDTE-S value since the FDTE-S metric is taken just before the frame submission call is made to the Xserver. We examine these results for all 3 types of scheduling semantics under this metric.



**Figure 5.10.** FDTE-C under Self Timed Control with High Load (FDTEC-ST-HL) (A)High Level View (HLV) (B) Detailed View(DV)

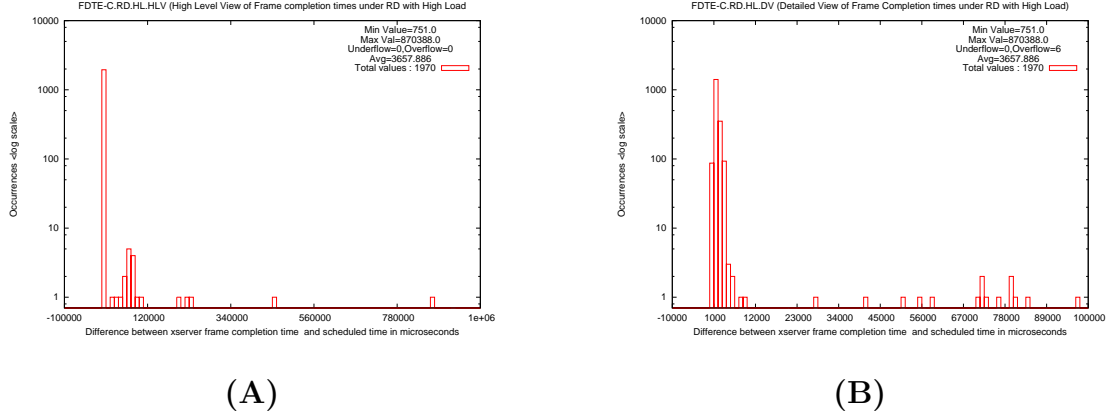
### Self Timed Control(ST)

The entire data range has been depicted in the Figure 5.10(A) which shows that all except 33 values are being distributed across the histogram range of 3171 microseconds to 1 second. This shows that under heavy load, there is a lot of contention in the scheduler for processing time due to which the Linux Xserver thread is being delayed in its display of the frame sent from the Display thread.

In the detailed view shown in Figure 5.10(B), we can see that 566 values overflow the range of the histogram. This is a clear indication that the performance of the ST under high load is poor and with a high amount of contention for the processor the Linux Xserver thread is not being scheduled as precisely as the application would like. Since more than 25 percent of the values are over 100 milliseconds (100000 microseconds) the user would be able to perceive the delays with ease and this is not good for any video application. This is another case of the performance of the ST application being poor when the system is under high stress.



## Read and Display thread Control(RD)



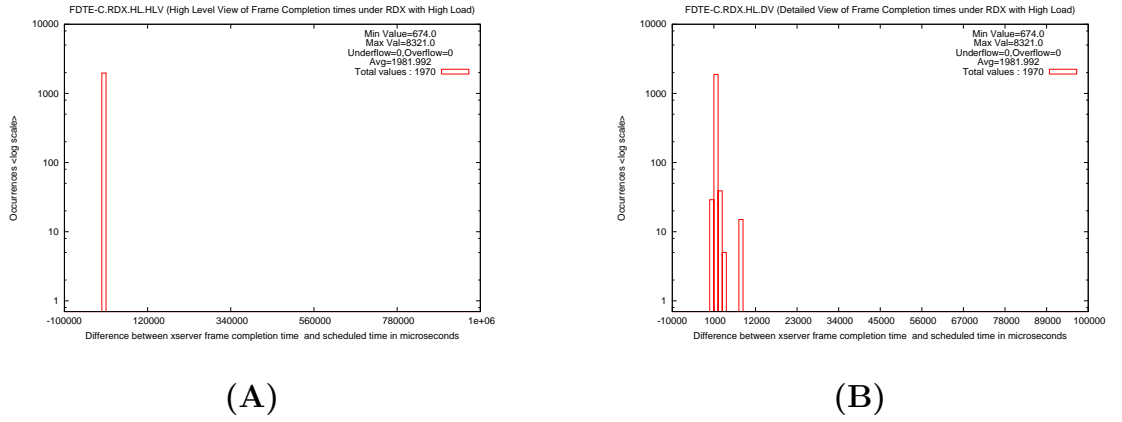
**Figure 5.11.** FDTE-C under Read and Display thread Control with High Load (FDTEC-RD-HL) (A)High Level View(HLV) (B) Detailed View(DV)

In Figure 5.11(A) we can see that most of the values are distributed between 1000 and 10000 microseconds. However, there are also other values that fall outside this range and in fact the maximum value of this histogram is 870 milliseconds which is almost a second of delay. There are 10 values falling outside the basic range in the histogram which will mean that there will be some glitches in the video but its performance is not as bad as ST under high load.

Figure 5.11(B) gives a more detailed look at the data. From this we can see that the values are uniformly distributed between 1000 and 10000 microseconds which is due to the variation in the standard Linux scheduler choosing the Xserver which is not under the control of the RD SDF. Also, there are overflows in the histogram whose maximum value is 100000 microseconds (100 milliseconds). These overflows are also due to the contention for the processor that exists under heavy load which in turn induces these delays in scheduling the Xserver. This gives ample reason

for us to take advantage of the SCPM to control the Xserver. And as shown in the description of Figure 5.12, the performance of the experiment with the Xserver thread under control is much better and produces a tighter distribution.

## Read, Display and Xserver Control(RDX)



**Figure 5.12.** FDTE-C under Read, Display and Xserver thread Control with High Load (FDTEC-RDX-HL) (A)High Level View(HLV) (B) Detailed View(DV)

Figure 5.12(A) illustrates the whole data range of the histogram which shows that all the values are clustered in the lowest positive range of the histogram. This graph looks identical to the FDTE-C under Light Load shown in Figure 5.6(A) which also had all the values in one bucket. This indicates that the performance of RDX Control is similar under both light and high load.

From the graph's detailed view in Figure 5.12(B), we can see that even under high load the distribution of the delay values is extremely tight and more than 99 percent of the values span from 674 microseconds to 2000 microseconds which is what was attained with the same experiment under light load. This proves that even though there is high load on the system, since the Group Scheduler takes

precedence over the Standard Linux scheduler we can see better control of the application with the RDX scheduler than the RD scheduler. That the Xserver was brought under Group Scheduling control has proved to be an advantage as we can see the difference between the RDX scheduler's results and RD scheduler's results clearly. When comparing this graph with those in Figure 5.11(B) in Page 77 and Figure 5.10(B) in Page 76, we can see that the number of values with high delays have reduced to become zero in the RDX under high load case whereas the RD under high load case had 10 values with big delays and the ST under high load case had almost 25 percent of the values with a delay of more than 100 milliseconds.

These comparisons support the argument that with these kinds of customized programming model we can definitely implement greater control of applications requiring them. Also, even though the increase in performance of RDX over RD is not extremely high, we can still say that in more computationally intensive applications where quicker response time from the system is required, a customized programming model that will control not only the application threads but also Linux system components is likely to provide greater performance.

## 5.2 Transformation Pipeline Application

This section discusses the experimental results for the Transformation Pipeline application that was discussed in Section 4.1.2. The semantic requirements of this application are similar to that of the Timed Playback and thus the same SDF that was explained in Section 3.3. The main difference in this experiment from the Timed Playback experiment is the addition of one pipeline video processing stage in between the "Read from File" thread and the Display thread.

This experiment was executed under both Lightly and Highly Loaded system conditions and tested with all the 3 schedulers that was discussed in Section 4.2. The notations of the RD and the RDX schedulers when discussing the results have changed to RTD and RTDX. RTD is Read, Transform and Display threads control and RTDX is Read, Transform, Display and Xserver threads control. Performance is measured using the FDTE-S and FDTE-C metrics for all the schedulers and under both the system conditions.

### 5.2.1 Lightly Loaded Conditions

This section examines the results of the Transformation Pipeline application when the system is running just the normal background system service and desktop application processes.

The results of the Transformation Pipeline application under light load was essentially identical to the Timed Playback application results under all the 3 schedulers. This is because the Transformation Pipeline also uses the same SDF's and has the same semantic constraints as the Timed Playback. The only noticeable difference in the results of all the experiments in this experiment is that the FDTE metrics of FDTE-S and FDTE-C have an extra delay due to the introduction of one extra thread of execution before the Display thread receives the frame to display. This delay does not affect the perception of the frames by the user viewing the video and is in the order of 600 microseconds. As with the Timed Playback, the performances of the RTD and RTDX schedulers have a considerably tighter distribution of frame delay values in the histogram than the Self Timed Control. The results under light load on system does not provide a lot of information about the performance of the SCPM since the standard Linux scheduler

would perform as well as GS since the competition for the CPU is low. So, for these reasons the results of the Transformation Pipeline application under light load for the 3 schedulers are not shown.

### **5.2.2 Highly Loaded Conditions**

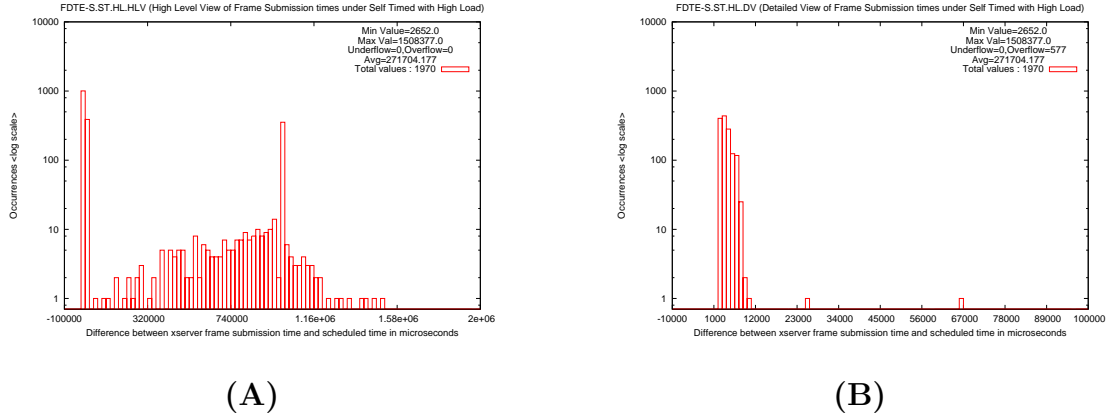
In this section we examine the results of the Transformation Pipeline application running with high load on the system. The load characteristics on the system are high CPU utilization processes, network interrupts and disk interrupts along with the basic system services. This load was created by 2 continuous Kernel compilations; one using the local disk and the other using a remote NFS file system, and the secure copy (scp) of a remote directory over the network.

#### **5.2.2.1 Frame Display Timing Error - Submission (FDTE-S)**

The metric used in these set of results are FDTE-S which is a best case approximation of the ideal FDTE metric. We examine the results for all the 3 types of scheduling semantics under this metric.

#### **Self Timed Control (ST)**

Figure 5.13(A) shows a histogram of the entire data set. This figure shows that all the FDTE-S values are distributed across the histogram range from 2652 microseconds to 1.5 seconds. This is extremely poor performance when compared to the Transformation Pipeline application requirements since almost 30 percent of the values are outliers to the first 2 positive buckets in the histogram. This would mean that once every four frame the video will become inconsistent with the Video frame schedule provided by the user. Also, since this ranges all the way



**Figure 5.13.** FDTE-S under Self Timed with High Load (FDTE-S-ST-HL) (A)High Level View(HLV) (B) Detailed View(DV)

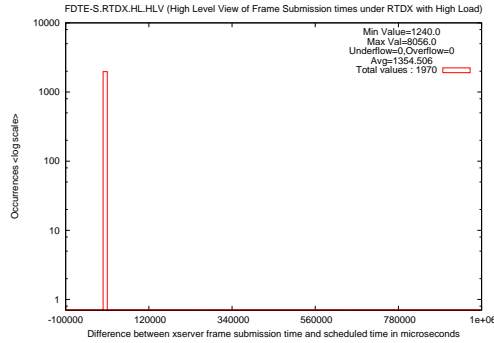
to 1.5 seconds the user will be seeing random stops in video frame display.

Figure 5.13(B) shows a detailed view of the data set. Notice that almost 30 percent of the values are overflowing the high range of 100 milliseconds (100000 microseconds) for this histogram. For the rest of the values, they are distributed between 2652 microseconds and 12000 microseconds as we saw in the Figure 5.13(B). These values are a result of the ST with SCPM not able to handle the Transformation Pipeline due to significant competition for the CPU by the many processes running in the system.

That so many histogram overflows have occurred in this experiment supports our argument that in a system that is under duress, the ST with SCPM does not perform well. When we compare the results of this experiment with that of RTD and RTDX schedulers under high load, it will be clear that the SCPM is better than that of the ST Programming Model. Recall that the performance of ST under high load was poor for the Timed Playback application as well.

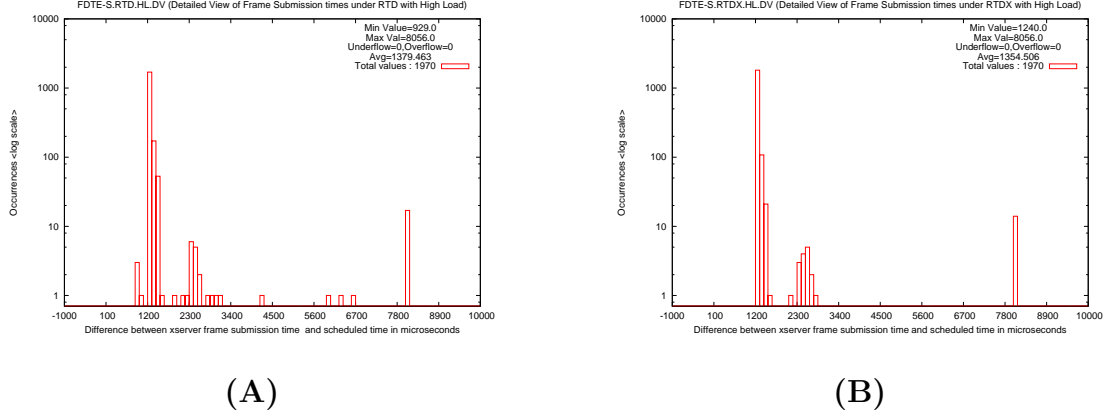
## RTD and RTDX Control

In this section we provide the results for both the RTD and the RTDX Schedulers put together because overall their performance is the same for both SDF's under FDTE-S and similar to the Timed Playback experiment. Also that the High Level View of the RTD and RTDX schedulers for this metric under High Load looks identical. Also, the reasoning that would hold for RTD is a subset of what will hold for RTDX since the FDTE-S metric does not involve the control of the Xserver thread which is the difference between the RTD and RTDX schedulers.



**Figure 5.14.** High Level View(HLV) of FDTE-S under RTDX Control with High Load (FDTE-S-RTDX-HL)

Figure 5.14 shows a histogram of the entire data set for the RTDX scheduler. Note that all the values are clustered in the lowest bucket of values. The histogram is identical to that of FDTE-S-RTD-HL for the Transformation Pipeline since the FDTE-S measures the submission of a frame and Xserver scheduling does not matter when this metric is measured. It shows that with RTDX and RTD control the performance of the system is significantly better than the ST control under High Load. The reason being that GS takes precedence over the Linux Scheduler and schedules the RTDX threads without interferences from other Linux processes.



**Figure 5.15.** Detailed View of FDTE-S with High Load under (A)RTD Control (FDTE-S-RTD-HL) (B) RTDX Control (FDTE-S-RTDX-HL)

In Figure 5.15, we show the detailed views of the histograms of RTD and RTDX schedulers when the system is under high load. We explain each of them below.

Figure 5.15(A) shows the detailed view of the data set under the RTD scheduler. The distribution of FDTE-S values in this histogram is such that the values of more than 95 percent of the 1970 frames fall between the minimum value of 929 microseconds and 2200 microseconds and the remaining 5 percent of the frames get displayed with delays between 2200 microseconds and 8056 microseconds. This is a delay that is not visible to the human eye. Comparing with the Figure 5.13(B), we can deduce that the performance numbers of FDTE-S-RTD-HL is far better than the same experiment under ST control.

Figure 5.15(B) shows the detailed view of the data set under the RTDX scheduler. Again, more than 95 percent of the values fall within the minimum value of 1240 microseconds and 2200 microseconds. There is a very small difference in the values of frame display delay in this histogram and Figure 5.15(A) as the FDTE-S

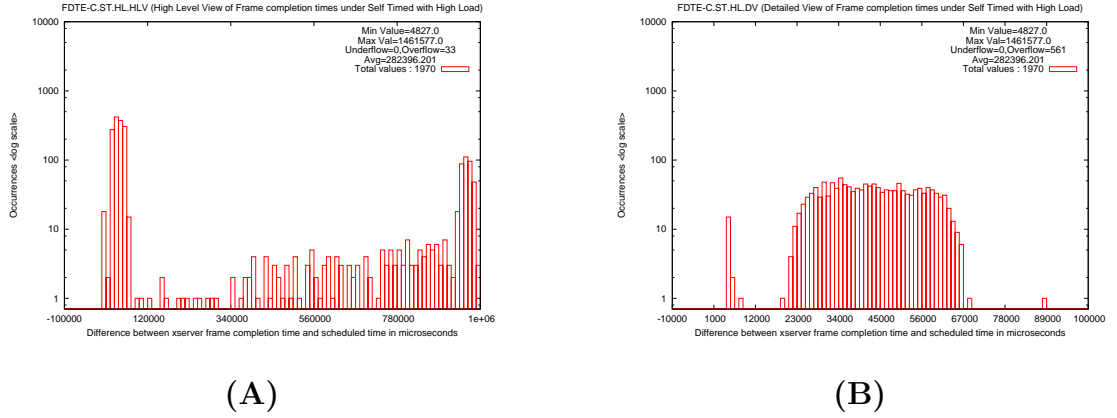


metric is calculated from the event that happened just before the frame submission to the Xserver and this means that the aspect of RTDX that is different from RD is not going to affect the outcome of the results measured with this metric.

### 5.2.2.2 Frame Display Timing Error - Completion (FDTE-C)

The metric used in these results are FDTE-C which is a worst case approximation of the ideal FDTE metric. We examine these results for all the 3 types of scheduling semantics under this metric.

#### Self Timed Control (ST)

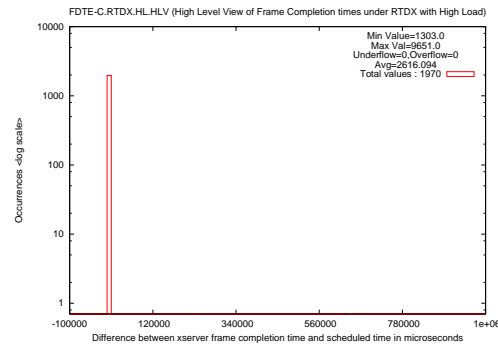


**Figure 5.16.** FDTE-C under Self Timed with High Load (FDTEC-ST-HL) (A)High Level View(HLV) (B) Detailed View(DV)

The entire data range has been depicted in the Figure 5.16(A) which shows that all except 33 values are being distributed across the histogram range of 4827 microseconds to 1 second. This shows that under heavy load, there is a huge amount of contention for the processor due to which the Xserver thread is being delayed for it to display the frame sent from the application.

In the detailed view shown in Figure 5.16(B), we can see that 561 values overflow the range of the histogram. This is a clear indication that the performance of the Self Timed under heavy load is poor and with a high amount of contention for the processor the Xserver thread is not being scheduled as the application would like it to be. Since more than 25 percent of the values are over 100 milliseconds (100000 microseconds) the user would be able to perceive the delays with ease and this is not good for any video application. This is another case of the performance of the ST application being poor when the system is under high stress.

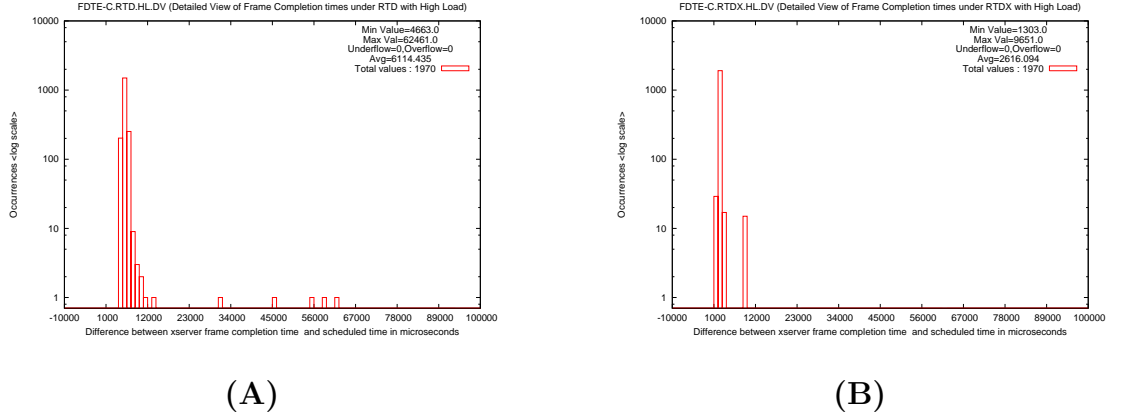
## RTD and RTDX Control



**Figure 5.17.** High Level View(HLV) of FDTE-C under RTDX Control with High Load (FDTEC-RTDX-HL)

Figure 5.17 illustrates the whole range of the data set which shows that most the values are clustered in the lowest bucket of values of the histogram. This graph looks identical to the FDTE-S under Heavy Load shown in Figure 5.14 which also had all the values in one bucket. This indicates that the performance of RTDX control does not degrade even when the worst case approximation measure of FDTE-C is measured. This is because the SDF controls the Xserver thread which

gets a chance to run as soon as the frame has been sent to the Xserver from the Transformation Pipeline application.



**Figure 5.18.** Detailed View of FDTE-C with High Load under (A)RTD Control (FDTEC-RTD-HL) (B) RTDX Control (FDTEC-RTDX-HL)

Figure 5.18(A) gives a detailed view of the histogram for the Transformation Pipeline application under RTD scheduler with high load for the FDTE-C metric. From this we can see that more than 95 percent of the frame delay values are distributed between 4663 microseconds and 10000 microseconds, which is proof of excellent performance of the experiment in comparison to its requirements of precise video playback. From the histogram, you will also find that there are some values which are scattered between 30000 and 62461 microseconds, which is the maximum value of the histogram. The reason for this is the contention for the processor that exists under heavy load which in turn induces these delays in scheduling the Xserver thread. This gives ample reason for us to take advantage of the SCPM to control the Xserver.

From the graph's detailed view in Figure 5.18(B), we can see that even under high load the distribution of the delay values is extremely tight and more than 95

percent of the delay values span from 1303 microseconds to 3400 microseconds. This proves that since GS takes precedence over the standard Linux scheduler we can see better control of the application with the RTDX scheduler than the RTD scheduler. That the Xserver was brought under GS control has proved to be an advantage that we can clearly see difference when comparing this graph with those in Figure 5.18(A) and Figure 5.16(B), we can see that the number of values with delays of more than 10000 microseconds have reduced to zero in the RTDX with high load case whereas the RTD with high load had 10 values with big delays and the ST with high load case had more than 25 percent of the values with a delay of more than 100 milliseconds.

## Chapter 6

# Conclusions and Future Work

The standard Linux scheduler's priority based scheduling does not provide reliable and precise response times for applications whose semantics cannot be expressed as priorities alone. So, in this thesis work we proposed a Customized Programming Model, the SCPM, that would solve this problem. The SCPM allows the application programmer to express applications semantics as a scheduling algorithm to Group Scheduling, which is a flexibly configurable scheduling framework. Using this method, the SCPM exhibits precise control over all the computations it is controlling as the scheduler takes decisions based on the real time information about the threads and in keeping with the application's semantics. Even in the face of high load, the application under SCPM control does remarkably better than when the application is under the Standard Linux based Programming Model.

The SCPM described in this thesis proved its efficiency using experiments that required the system to precisely schedule its threads in order to follow a pre-determined schedule. As future work, an application that can simulate the arrival of packets of data on the network can be precisely controlled using our

SCPM. This application can be designed to have a thread that would act as a simulator of network arrival that would control the propagation of data in the pipeline according to a pre-determined schedule. A modification to the Transformation Pipeline application discussed in Section 5.2 will be sufficient to do this future work. Using this application, testing of network related applications can be achieved with higher precision.

Video related applications require precise real time computation control. The driving application used in this thesis work also used video related applications and demonstrated the efficiency of the SCPM. GStreamer [3] is a video framework that facilitates the creation of media applications. GStreamer also uses a pipeline model to stream video through an application. Since our programming model can control pipeline application efficiently, it is conceivable that the SCPM can be used to control GStreamer threads. The main challenge would be to research the thread model that GStreamer uses and whether that thread model will be able to map on to the SCPM.

# References

- [1] K. Flanigan. Utilizing Shared Memory across the Kernel Userspace Boundary to Improve the Efficiency of Precise Computation Control. Undergraduate Honours Thesis. April 2006.
- [2] M. Frisbie, D. Niehaus, V. Subramonian, and Christopher Gill. Group scheduling in systems software. In *Workshop on Parallel and Distributed Real-Time Systems*, Santa Fe, NM, apr 2004.
- [3] GStreamer. <http://gstreamer.freedesktop.org/>.
- [4] Linux Weekly News. <http://lwn.net/Articles/199643/>.
- [5] S. Rostedt and Darren V. Hart. Internals of the RT Patch. In *Ottawa Linux Symposium*, June 2007.
- [6] Douglas Niehaus Tejasvi Aswathanarayana, Venkita Subramonian and Christopher Gill. Design and Performance of Configurable Endsystem Scheduling Mechanisms. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2005)*, May 2005.
- [7] Wikipedia. [http://en.wikipedia.org/wiki/Completely\\_Fair\\_Scheduler](http://en.wikipedia.org/wiki/Completely_Fair_Scheduler).

- [8] Edward A. Lee Yang Zhao, Jie Liu. A programming model for time-synchronized distributed real-time systems. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 07)*, Bellevue, WA, April 2007.